University of Alberta

Library Release Form

Name of Author: Rohit V. Kapoor

Title of Thesis: Mathaino: Device-Retargetable User Interface Reengineering Using XML

Degree: Master of Science

Year this Degree Granted: 2001

University of Alberta

Device-Retargetable User Interface Reengineering Using XML

by

Rohit V. Kapoor ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements of the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2001

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Device-Retargetable User Interface Reengineering Using XML** submitted by Rohit V. Kapoor in partial fulfillment of the requirements for the degree of **Master of Science**.

To my son Rishi, Erica, my parents and my sister Gunjan.

Abstract

With the increasing proliferation of computing devices and platforms, it has become increasingly important for organizations to make their existing software systems accessible to new environments. However, almost all the proposed solutions for this problem focus either on constructing user interface translators for the legacy system or on completely migrating the existing legacy system or its database to a new platform.

Mathaino is a platform independent, non-invasive system that is capable of simultaneously migrating text based legacy systems to a range of modern GUI based target platforms. Unlike previously proposed solutions, which are semiautomatic at the best and deal with only a single target platform, the goals of Mathaino are to further automate the process of legacy system migration and enable simultaneous migration to multiple platforms.

Acknowledgements

I would like to express my deep gratitude towards my supervisor, Dr. Eleni Stroulia, for her supervision, encouragement and motivation through the course of this work. Without her vision, guidance, and insightful comments, this work would not have been possible.

I am also grateful to members of CelLEST project Roland Penner, Paul Iglinski, Mohammad El-Ramly, Vijayan Menon and Huaxin Zhang for their cooperation and suggestions throughout the tenure of this project.

Finally, I would like to thank Erica, my parents and my son Rishi for their patience, support and continuous encouragement.

Rohit V. Kapoor

June, 2001.

Edmonton, Canada.

Table of Contents

Chapter 1

Introduction

Legacy information systems (LIS) can broadly be defined as computer software systems, which despite being technologically antiquated, continue to be extremely important to an organization. As more and more organizations move to delivering their services on the Web, the migration of the interfaces of such systems to the Web is becoming a major activity in the Information Technology field and an important research problem in software reengineering.

In general, software-engineering research on system migration has followed one of three dimensions. First, several methodologies have been proposed for making the legacy maintenance and evolution practices more principled, measurable and controllable [WU97, LEHMAN98]. A second line of research has focused on the problem of code understanding for the purpose of isolating components of the legacy system so that they can be reused in a new environment [SNEED92]. Finally, a third line of investigation has focused on the problem of migrating text-based user interfaces to GUI environments, by inferring the widgets implied in the interaction provided by the original interface and

using knowledge-based methods to translate them to corresponding widgets in the target environment [MOORE94].

Code understanding can be extremely complex and error-prone task; *widget translation* has to be repeated every time the legacy system needs to be made available on a new platform. In our work with the Mathaino system, we have adopted a novel approach to this problem. Our objective is to develop an intelligent system that can model the current interface of a legacy system at the user-interaction level and design a functionally equivalent interface on another platform.

Mathaino's model of the system-user interaction through the current system interface is learned from examples of expert system users performing several instances of the same task. Mathaino generalizes the observed user behavior and constructs a model of the user navigation through the system interface and the information exchange between the system and the user. This abstract model constitutes a plan for the task-specific system-user interaction and specifies the information input by and provided to the user and the elementary interactions on the current interface that enable this information exchange. This abstract plan can be executed by Mathaino's run-time environment, on various platforms by exposing a simpler and optimized interaction to the user. Mathaino's run-time environment, given a task-specific interaction plan, constitutes a task-specific, but platform-independent wrapper of the legacy system. This wrapper makes it possible to access the legacy system from a variety of platforms, and thus address the legacy migration problem.

1.1 What is a Legacy System?

Historically there have been many definitions of what exactly constitutes a legacy system. So instead of inventing yet another new definition for legacy systems we list and critique the important ones below:

"Large software systems that we don't know how to cope with but that are vital to our organization" [BENN95].

This definition isn't a very accurate one as it classifies any large computer software system that has not been documented properly as a legacy system. Thus, according to this definition even a large, new, albeit poorly documented software system should be classified as a legacy system.

"A computer system or application program which continues to be used because of the prohibitive cost of replacing or redesigning it and despite its poor competitiveness and compatibility with modern equivalents. The implication is that the system is large, monolithic and difficult to modify" [FOLD96].

This is not a very accurate definition for a legacy system either as usually a legacy system is far from being non-competitive. In fact, the very reason for its continued existence is that it remains a competitive solution for the problem that it solves.

"Any information system that significantly resists modification and evolution to meet new and constantly changing business requirements" [PERR95].

"Legacy Software is critical software that cannot be modified efficiently" [GOLD98].

Again, according to these definitions even brand new software systems that resist modification or evolution are legacy systems. While this might be an interesting point of view, it is not a very accurate definition of a legacy system. This is because even if a piece of software can be modified or evolved easily it can still be classified as a legacy system as it may have been built using legacy technologies or it might work only on legacy hardware.

Thus, there is no one absolutely accurate definition for legacy systems in the current literature. However, usually legacy systems can be easily identified by the various unique characteristics that they exhibit. Thus, a more accurate approach for identifying a legacy system is to check whether it is plagued by these typical problems associated with legacy systems. Some of the important traits of legacy systems are listed below:

1. Legacy information systems usually run on antiquated hardware.
2. They actively resist attempts at modification and evolution.
3. They are poorly documented and understood.

4. They lack a clean architecture and interfaces making their integration with other information systems difficult.
5. They usually form the core of their organization's IT infrastructure.
6. They encapsulate within themselves important business rules and store vital business data accumulated over years of operation.

1.2 Scope of the Legacy System Migration Problem

Due to the problems highlighted in the previous section, migration of any legacy information system is a vital but highly risk prone task for an organization. Also as the field of information technology matures, the proportion of legacy systems in use will increase affecting more and more organizations further exasperating this migration problem.

Legacy code has been compared to the Chernobyl nuclear disaster - *too messy to clean up but too dangerous to ignore* [PHNXGRP]. Even at present the statistics of the LIS migration problem are stunning. Some of the examples of the scale of this problem are discussed below.

Despite all the hype about new object oriented languages like Java, C++ etc. when it comes to real-life business systems legacy code and legacy applications still rule the roost. According to a Gartner Group study currently there are about 200 billion lines of COBOL legacy code, still growing at about 5 billion lines a year. Corporations have

about 5 trillion dollars invested in these business systems. There are approximately 2.4 million COBOL programmers, maintaining and writing over 9.5 million COBOL applications. This represents over 60% of the world's total computer code. Since all these applications exist and are being used even today the market for legacy system maintenance and migration is phenomenal [CARR00, HILL01].

The year 2000 problem, popularly known as the Y2K problem, was essentially a legacy system migration problem. Almost all banks, financial institutions, hospitals, power plants, aerospace infrastructure like aircraft, airports etc. were affected by this problem [BISB97].

A US Department of Defense (DOD) study indicated that within DOD itself there are at least 1.4 billion lines of legacy code that annually consume a large part of DOD's IT budget of about 9 billion dollars to maintain [AIKE94].

Legacy COBOL business applications have proved to be so difficult to migrate that even now the only efficient means of maintaining and modifying them is via using the old COBOL language. Recently, Fujitsu has been trying to migrate the COBOL language itself to a web enabled platform so that these legacy COBOL applications can be migrated to the web [FUJITSU01, ARRA00].

While a satisfactory analysis of the scale of the legacy system problem does not exist, the fact that many large computer corporations derive a large chunk of their revenues from

legacy system solutions points to the fact that legacy system maintenance and migration is a multi-billion dollar industry. For example, the IBM corporation provides a whole set of software solutions for legacy systems, simply because most of the legacy systems run on IBM hardware and software [PHNXGRP] and has large projects like the San Francisco project which provides a standard way of accessing legacy data via an AS/400.

Perhaps the most important factor affecting the LIS migration problem is that it will always be there. Even state of the art software systems of today will elapse into legacy systems of tomorrow.

1.3 Issues in Legacy System Migration and Interoperation

As already pointed out, legacy COBOL code itself forms about 60% of all the computer code in the world today. Thus, a big majority of business applications operating today are legacy applications.

Increasingly for today's businesses, moving their legacy applications to the web is becoming a major problem. Also another major issue is the integration of existing legacy applications with new applications and emerging technologies. It is not uncommon to find multiple versions of the same application existing within an organization. While the newer version is used for servicing today's needs, the old legacy application is also retained while a viable base of old users (customers) exists for it. In this scenario it becomes very important that the old and new versions of the applications are able to

integrate otherwise problems associated with multiple databases like data duplication and data consistency arise.

1.3.1 Classification of Legacy Systems

A legacy system can be classified into three major categories [WEID97]:

1. Healthy
2. Ill
3. Terminally Ill

A legacy system that is presently in harmony with the current needs of its organization or needs only minor updates to keep up with the change in needs can be classified as a healthy legacy system. A legacy system can be categorized as ill if it was either constructed with a poor architecture and design, was maintained improperly or if the needs of its organization have changed so drastically that it can't cope up with them easily. A terminally ill legacy system is one where it is more beneficial to replace it completely rather then try and maintain it.

1.3.2 Choosing the Appropriate Migration Technique

There are multiple techniques for legacy system migration [BISB99].

In this section we discuss the broad categories of various legacy system migration techniques and dwell upon the issues concerning the choice of the appropriate migration technique to use for a particular legacy information system. We discuss the specific examples of these broad categories in section 1.4

Broadly speaking, we can maintain, wrap or redevelop an existing legacy information system. Again each one of these techniques can be sub-divided further. For example, wrapping can be done in a wide variety of ways. We can wrap individual modules of the legacy system (white box wrapping), wrap only the legacy database (database migration), or use screen scrapping [CARR98] (black box wrapping). Again, maintenance can be preventive, adaptive, corrective or perfective. Figure 1.1 shows these legacy system migration solutions and their impact on the legacy system.

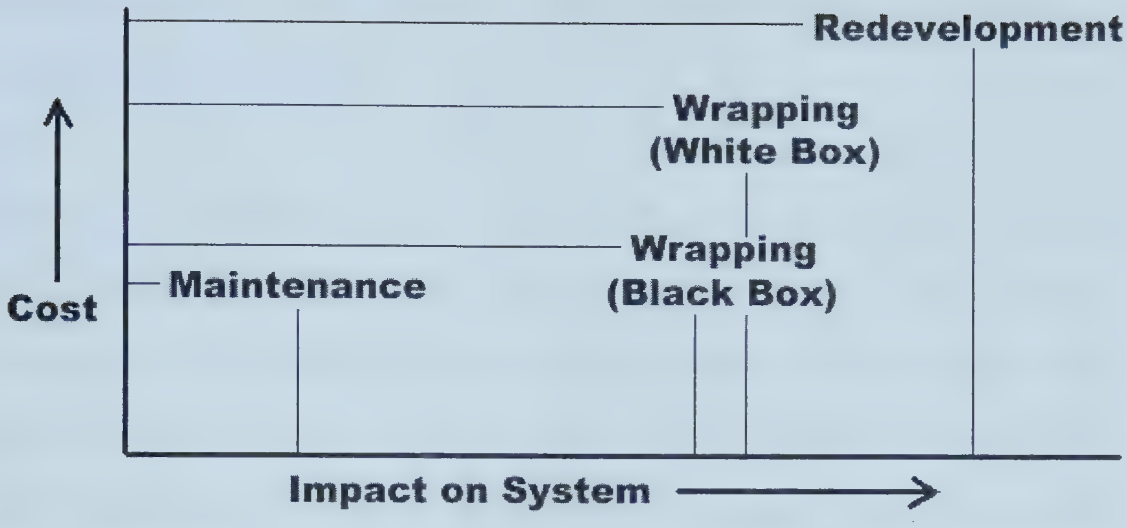


Figure 1.1: Cost Vs Impact of Different Migration Techniques

Redevelopment, also sometimes referred to as the Chicken Little or Big Bang Approach [BISB99], is a highly cost and risk intensive strategy as here the main aim is to redevelop the legacy system from scratch using modern architecture, tools, and databases, on a modern hardware. This approach is not a cure-all as the newly developed system will itself evolve into another legacy system with time, also the risk of failure is usually unacceptable. However this is the advisable approach to take when the system is terminally ill.

Wrapping or black box migration is a very generic term that within it encompasses a wide variety of techniques that can be used to migrate a legacy system. Almost any wrapping solution has a risk factor lower than complete redevelopment and higher than routine maintenance. Further, wrapping a legacy system is usually a much lower cost process than complete redevelopment. Also, if wrapping is done automatically then at least some wrapping solutions (black box wrapping) can cost as low as, if not lower, than

routine maintenance. Currently the most popular automated LIS wrapping solution in existence is screen scrapping.

Depending on its situation any one of these techniques can be used for migrating a legacy system. The technique to be used for migrating an existing legacy system is largely determined by its current state of health. While it is certainly inadvisable to even try and retrofit a terminally ill legacy system, it is better to try and update an ill legacy system rather than completely replace it. While migration techniques like wrapping don't increase the performance of the underlying legacy system or improve its design they can still be very useful for healthy or ill legacy systems where the underlying legacy system is still performing satisfactorily. In these cases the lost cost and low risk of wrapping based migration techniques make them a very attractive option. Thus, the cost and risk ratios are the prime decision factors governing the future of a legacy system in this scheme.

Another very interesting metrics based approach for deciding the category of reengineering needed for a particular legacy system was introduced by Harry Sneed in [SNEED95]. According to this approach, we should be able to quantitatively identify the class of legacy systems that require aggressive reengineering, from those that should be reengineered on a lower priority and those that should be completely redeveloped. At the heart of this approach lies a quality Vs business value graph (shown in Figure 1.2, for a discussion about calculating the appropriate bounds for this graph and the associated metrics please refer to [SNEED95]), henceforth referred to as the Sneed graph.

According to Sneed, the reengineering resources must be prioritized to those legacy systems that are of high business value to the organization, while those legacy systems that are of lower value could either be redeveloped or outsourced since they do not pose a significant business risk. Thus, according to Sneed the business systems that lie on the lower right corner of the Sneed graph (refer to Figure 1.2) must be reengineered with high priority while those on the lower left corner should most probably be replaced. Those on the higher right corner may be reengineered with a low priority. Interestingly, the solution proposed by Sneed doesn't take the current health of a legacy system into account while proposing a migration path for it.

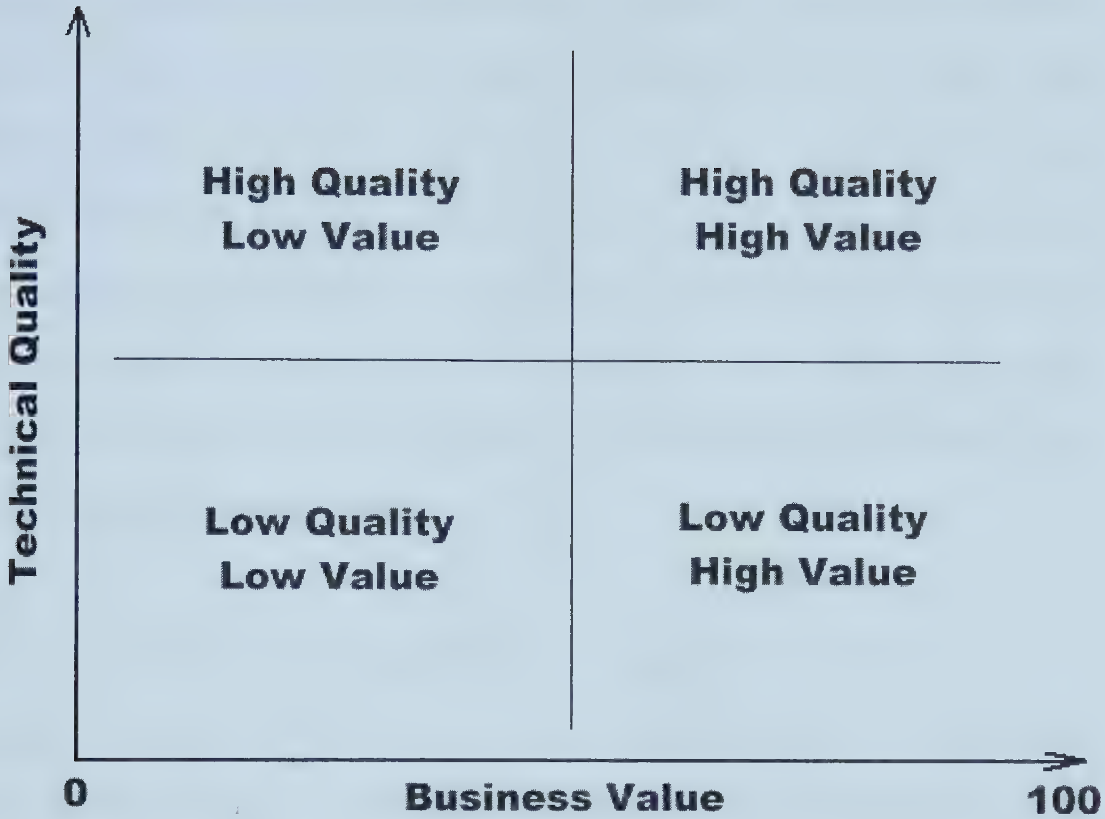


Figure 1.2: Sneed's Graph for Planning the Appropriate Reengineering Solution From [SNEED95:29]

Anyhow, the Sneed approach again highlights the fact that cost of migration and the risk associated with it do play a major role in deciding the future of any particular legacy information system. For example, if a low cost and risk free solution for migrating the entire spectrum of legacy information systems existed then most of these calculations would become redundant.

Also it has been found that the cost/benefit ratio of white box migration techniques have more or less stayed constant over time, while the cost/benefit ratio for black box

migration techniques have dropped considerably [WEID97]. Coupled along with this measurement is the risk factor. While white box techniques for migrating these legacy applications have usually achieved only a limited success and represent a very risk intensive path for system migration, black box migration techniques have evolved to a point where at least some of them (e.g. screen wrapping) represent a very low to almost zero risk path for migrating the existing legacy system. Also as many of these legacy applications still provide acceptable performance, the main task is not redevelopment but system migration and inter-operation.

1.3.3 Migration issues arising from the emergence of the Internet

As stated previously, another big factor encouraging the migration of even healthy legacy systems is the emergence of the Internet. These days migration of legacy systems to the World Wide Web is becoming increasingly important. However, migration of a legacy system to the Internet is not simply a matter of producing an HTML interface for it. Currently a wide variety of devices can connect to the Internet. While the HTML platform may be sufficient for simplistic case of web browsers running on PCs, increasingly the users of an online software system are demanding that their systems be available on the entire spectrum of devices that they can use to connect to the Internet. Since, other online platforms like the PDAs etc. work on interfaces other than HTML (for example the mobile internet devices usually work on WML), migrating an existing legacy system to HTML is not a sufficient solution anymore. Furthermore, organizations are also interested in making their legacy applications available over custom platforms

like Java or C++ where, again, the end user needs to log into the legacy system over the Internet.

To overcome some of these migration problems, recently, XML wrapping has been proposed as an alternative to traditional legacy system migration techniques so that the legacy system's data can be exported to a wide variety of devices and platforms [COME01]. But even though exporting the messages and the database of a legacy system in the XML notation may make it available to a larger audience, the problem of migrating the entire legacy system along with its business logic to multiple platforms still remains.

Thus, while a lot of research has been done towards migrating legacy databases to the web [PERR95] and also simple user interface migration solutions like screen scrapping exist, none of these provides a satisfactory solution for the entire spectrum of the system migration problems. As discussed previously, even in the newer XML wrapping solution, the only new feature added is that the wrapped system's messages are exchanged in XML notation. While this makes the legacy system's information available to a larger community the process of wrapping the system is still quite rudimentary and limited to the database component of the legacy system. Also, post-1995 research and emergence of the Internet have proved that black box migration solutions are quite an attractive option for dealing with non-terminally ill legacy systems. However, a low cost, risk free, multiple platform, highly automated migration solution for an entire non-terminally ill legacy system still does not exist.

1.4 Existing Legacy System Migration Solutions

Legacy system migration has been a relatively hot research topic throughout the 1990s. A wide array of techniques has been proposed for effectively dealing with legacy systems. Some of the important ones among these are discussed in this section.

1.4.1 Maintenance

While maintenance may not lead to migration of the legacy system, it is still one of the most popular solutions for prolonging the life of legacy systems. Although, some authors differentiate between maintenance and re-engineering (see [SNEED95]), we regard reengineering as aggressive maintenance. Thus, we discuss reengineering projects within this section as the issues discussed in this section pertain to reengineering as much as they do to migration.

Effective legacy system maintenance has proved to be a hard problem. The primary causes of most maintenance headaches are:

1. **Poor or No Architecture**

   A majority of legacy systems are old systems that were designed in 1970s. As a consequence the most prevalent architecture pattern for legacy systems is the spaghetti code pattern. This lack of any formal architecture makes maintenance of legacy systems a difficult, costly and risk ridden process.

2. Lack of Documentation

Either legacy systems have no documentation or if it exists it is usually minimalist. Usually understanding the legacy code itself is ninety-percent job done. For example, the RT-1000 system that Nortel engineers had to maintain had absolutely no documentation and required one whole year of painstaking research and a fifty percent turnover rate to re-document [RUGA98]. This lack of documentation significantly increases the amount of resources required for maintaining a legacy system.

3. Language Soup

Usually a majority of research papers on legacy code understanding concentrate on analyzing the source code for a legacy system coded in a particular language. Further, almost all of these analysis tools are not very accurate as far as automatically generating artifacts like design, algorithm, architecture are concerned. However, even if they were 100% accurate they would still be inadequate as a majority of large legacy systems are coded using multiple languages. For example, Munson cites the example of a legacy system written in at least eight different programming languages [MUNSON98]. Again, one of the major problems faced by the RT-1000 team was that the system had been coded in both Fortran and C and use multiple processes resulting in the failure of the automated code analysis tool that they were experimenting with [RUGA98]. Thus, the language soup that constitutes any legacy system's code is another major source for severe migration headaches.

4. **Lack of Appropriate Skills**

   Legacy systems are coded in archaic languages, using archaic algorithms and techniques. It is quite difficult of find someone with the right skill set for maintaining them. Indeed, it has happened that the maintenance of a legacy system had to be frozen because the only person who has the right skills was no longer available. For example, for one legacy system the maintenance of one of its modules had to be frozen as it was coded in SNX and the only programmer who knew SNX in that organization died of a heart attack [MUNSON98]. Thus, lack of appropriate skills is also a major factor affecting maintenance.

Despite these problems maintenance retains its place as the most prevalent and cost intensive development activity for any software system. Further, research has shown that software systems built in accordance with modern software engineering techniques require more not less maintenance [GLASS98]. However, maintenance is only effective for healthy legacy systems that do not require any radical changes in their operation and is not a viable alternative in most other cases. It has also been observed that with time maintenance tends to decrease the overall quality of the software system till a point is reached when it is no longer beneficial to simply maintain it anymore and more radical solutions are needed. Thus, while maintenance is a popular solution its domain of application is restricted and it can not be treated as a final solution for any legacy system.

1.4.2 System Migration

System migration is the science of modifying an existing legacy system so that it can work with new languages, operating systems, and/or hardware. The prime concerns while attempting the migration of any legacy system are:

1. Most of its existing legacy code base must be retained.
2. Minimization of migration costs
3. Minimization of risks

As discussed before, system migration can be achieved by either modifying the source code (white box migration) or by wrapping it (black box migration).

Reverse engineering is the major theme in white box migration as here the main task is to examine and understand the existing legacy system [CHIK90]. Obviously, code analysis, design recovery and re-documentation tools play a major role in this effort. Some examples of successful white box migration projects are the Nortel RT-1000 Software System [RUGA98] and the configuration management system at Sandia National Laboratories [BRAY95]. However, even these two migration projects overshot their budget and consumed a lot of human expertise. Also, while a large number of code understanding tools have been proposed none of them provides a 100% reliable and accurate solution, and they all require a non-trivial amount of manual human intervention. Further, as pointed out by [WONG95] while almost all code understanding tools concentrate on documenting "in-the-small" i.e. documenting the data structures and

algorithms of a given LIS, at least for large systems what is actually needed for true understanding is documenting "in-the-large" i.e. documenting the higher-level structural aspects of the system like its architecture. None of the automated code-understanding tools are capable of this. Thus, while white box migration may be the only solution for a certain class of legacy systems, it requires substantial resources in the form of human expertise and budget to succeed. So a careful analysis and exhaustive research is needed before commencing on a white box migration project.

Black box migration has emerged as the most successful, automated, cost effective and low risk approach for dealing with legacy systems [WEID97]. A number of black box migration solutions have been proposed. Commercially, screen scrapping [PANGIA, CARR98] is the most effective and widely used black box migration technique. However, screen scrapping is a very coarse technique for migrating a legacy systems as it simply translates the vanilla 3270 text interface of a legacy system to simple text based HTML form. The resulting user interface is just as tough and user-unfriendly as the original text interface of the legacy system. Further, screen scrapping can work effectively only when the legacy system uses TCP-3270 as its communication protocol as it relies heavily on the extra field information that is provided by this protocol. Thus, for a large number of legacy systems (like VT100 system etc.) that do not support this protocol, screen scrapping remains an unavailable option.

Merlo et. al. have presented an alternate user interface re-engineering technique in their paper [MERLO95]. However, a closer inspection reveals that what they are proposing is

essentially a white box approach to screen scrapping, which not only has the same disadvantages as vanilla screen scrapping but also tags along the bag of problems associated with white box migration techniques.

Another commercially successful black box option for migrating legacy systems to the web is Batch Processing [PANGIA]. In batch processing the main aim is to make the legacy system's database available on the web. However, if the legacy database does not support technologies like SQL, ODBC etc. then usually a manual operator is needed for this scheme to work. Thus, it is a very fragile and manual solution for migrating a legacy system to the web.

Object wrapping [PHNXGRP] is the other proposed black box migration technique. Wrapping legacy systems using objects certainly has some important benefits. For example, a legacy system module that has been wrapped within an object can inter-operate or become a constituent of a new object oriented system architecture making that particular module more flexible. Currently, we can either wrap an entire application or individual components. However, finding the correct module to wrap can be a cumbersome process, specially considering the fact that legacy information systems were not designed in accordance with a clean architectural pattern and, thus, have highly a convoluted code base. Again, if the scope of wrapping is the entire application then the only feasible way to accomplish this at present is through screen scrapping techniques, whose disadvantages have already been discussed in the previous sections. However, if it was possible to decompose the domain of the existing legacy system into its constituent

objects and wrap the entire legacy system into modules consisting of these domain objects at the user interface level then the problem of decomposing the convoluted source code of the legacy system into discrete functional objects would disappear. This is one of the ideas that we have adopted in our implementation of Mathaino and it is discussed further in Chapter 3.

1.5 Conclusion

In the rest of this thesis we will discuss the various aspects of the Mathaino solution in detail. In Chapter 2 we will discuss the overall Mathaino architecture. In Chapters 3 and 4 we will discuss the development and runtime processes that are used to migrate an existing legacy system using Mathaino. In Chapter 5 we will discuss the software engineering aspect of Mathaino i.e. the methodology used for developing Mathaino. Finally in Chapter 6, we will highlight the differences between our approach and the previously proposed solutions in this domain and conclude.

Chapter 2

Architecture for Mathaino Developer and Developer

During the last decade of twentieth century, the emerging field of software architecture has been a major area of focus for the software engineering community. Mary Shaw, who is one of the principal researchers in the field of software architecture, defines it as:

"Software architecture involves the description of elements from which (software) systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns" [SHAW96].

The field of software architecture emerged and evolved as software engineers slowly realized over time that while almost any software system could be decomposed into its constituent subsystems, usually successful software solutions had typical organizational structures or idioms for their constituent subsystems that could be represented using more generic patterns. Over a period of time a multitude of architectural patterns have been identified [SHAW95]. Some of the more prevalent ones are listed below:

1. Pipeline Architecture
2. Model-View-Controller (MVC) Architecture
3. Data Abstraction
4. Repository Architecture
5. Stovepipe (or lack of) Architecture
6. Interpreter Architecture
7. Layered Architecture
8. N-tier Architecture

Furthermore, research in the field of software architecture has shown that these typical architectural patterns are not only useful for cleaner program design and organization but also each one of these architectural patterns is particularly suitable for software systems that solve a specific class of design problems [GAMMA93]. For example, a particularly useful and commercially popular architectural pattern is the Model-View-Controller (MVC) architectural pattern. This architecture is very well suited for graphical user interface intensive applications and frameworks as it separates the representation of data from its visual presentation, thus, allowing multiple graphical views to mirror the same data model. This significantly decreases the design complexity of such software systems. However, this specific architectural pattern is of little use in other situations where the prime objective of the software system is not presentation of data graphically. Thus, while this pattern is used intensively in commercial GUI design frameworks like Swing [ROBINSON99] and MFC [SHEPHERD96], it is of little use in other situations, say real-time applications.

Thus, the architectural design of an application largely depends upon the nature of problem that it is trying to solve [BECK94]. Also, initial work on software architecture concentrated on decomposing an entire software system into a single large architectural pattern. It is now known that this approach is too simplistic, that various architectural design patterns are not mutually exclusive. It is more appropriate to regard a large software system as an amalgamation of multiple design patterns each of which is well suited for at least one of the major problems that a given software system is trying to address.

2.1 Basic Mathaino Architecture

As pointed out in the previous section the appropriate set of architectural patterns for a software system can be derived from the set of primary specifications for that system. While designing Mathaino our prime concerns were:

1. As Mathaino was an active research project, it was obvious to us from the start that situations will arise where we would have to heavily modify entire sets of its subsystems while isolating the unaffected portions.

2. As research projects have a constantly changing set of requirements, it was necessary to have a highly extensible system architecture that would allow us to add new functionality relatively easily.

3. It was also necessary that Mathaino should be able to run on a multitude of platforms as legacy information systems exist on a wide range of hardware and software combinations.

To satisfy these design constraints we choose to implement Mathaino using a combination of two different architectural patterns:

1. The Layered Architectural Pattern
2. The Component Based Architectural Pattern

We decided to stick with the layered architectural design pattern as the primary design with the component based design being used for integrating the various components and layers. The layered architectural pattern was well suited to our constraints because it allows system decomposition into multiple isolated layers with the stable components usually forming the lower layers while the unstable components constituting the higher layers. Thus, the layered architectural pattern allowed us to satisfy the first constraint as we could readily move volatile components to the higher layers and modify them extensively while isolating the components in the other layers from any side effects. Further, the layered architectural pattern is also quite useful for progressively abstracting away various platform-oriented idiosyncrasies and isolating the higher layers from the actual implementation platform. This feature was particularly appealing to us as it helped us in satisfying our third architectural constraint.

As already discussed, we also required our implementation architecture to allows us the ability to add new functionality and capabilities to the existing system fairly easily without requiring significant modifications to other components. The component-based architectural pattern is well known for both its scalability and extensibility. Some of the features of software components are [DERI97]:

1. They have a well-defined interface that allows additional components to be developed and added to the existing set in a fairly ordered manner.
2. They are not statically linked to the application but are loaded at run time.
3. They have the ability to be replaced fairly easily, even at run time.

Thus, this architectural pattern was well suited to satisfy our second constraint as it allowed us to incorporate newer components into the existing framework to meet the constantly emerging requirements without modifying the underlying core application. Also as Mathaino is required to produce platform independent user interfaces, a component-based architecture is quite beneficial as it allows any application that follows Mathaino's plugin standards to interface with it.

Figure 2.1 illustrates the basic Mathaino architecture. Mathaino utilizes two Java Virtual Machines (Java 1 and Java 2) and the Java ODBC API in its base layer. We decided to build Mathaino on top of the Java platform as that allows us to simultaneously develop

Mathaino for a wide range of platforms helping us further in satisfying our third architectural constraint.

The Mathaino base system, which forms the next layer, is responsible for providing the basic plugin framework over which rest of the system is built. It is this layer that forms the heart of the Mathaino's component-based architectural pattern. The various Mathaino subsystems plug into this base system. This architecture makes Mathaino very extendable; thus, additional capabilities can be added to it very easily. Further, the use of the layered architectural pattern, coupled with the use of Java and XML makes the entire Mathaino system very portable. Currently, versions of Mathaino exist for Windows NT/98/95, LINUX, and Solaris platforms.

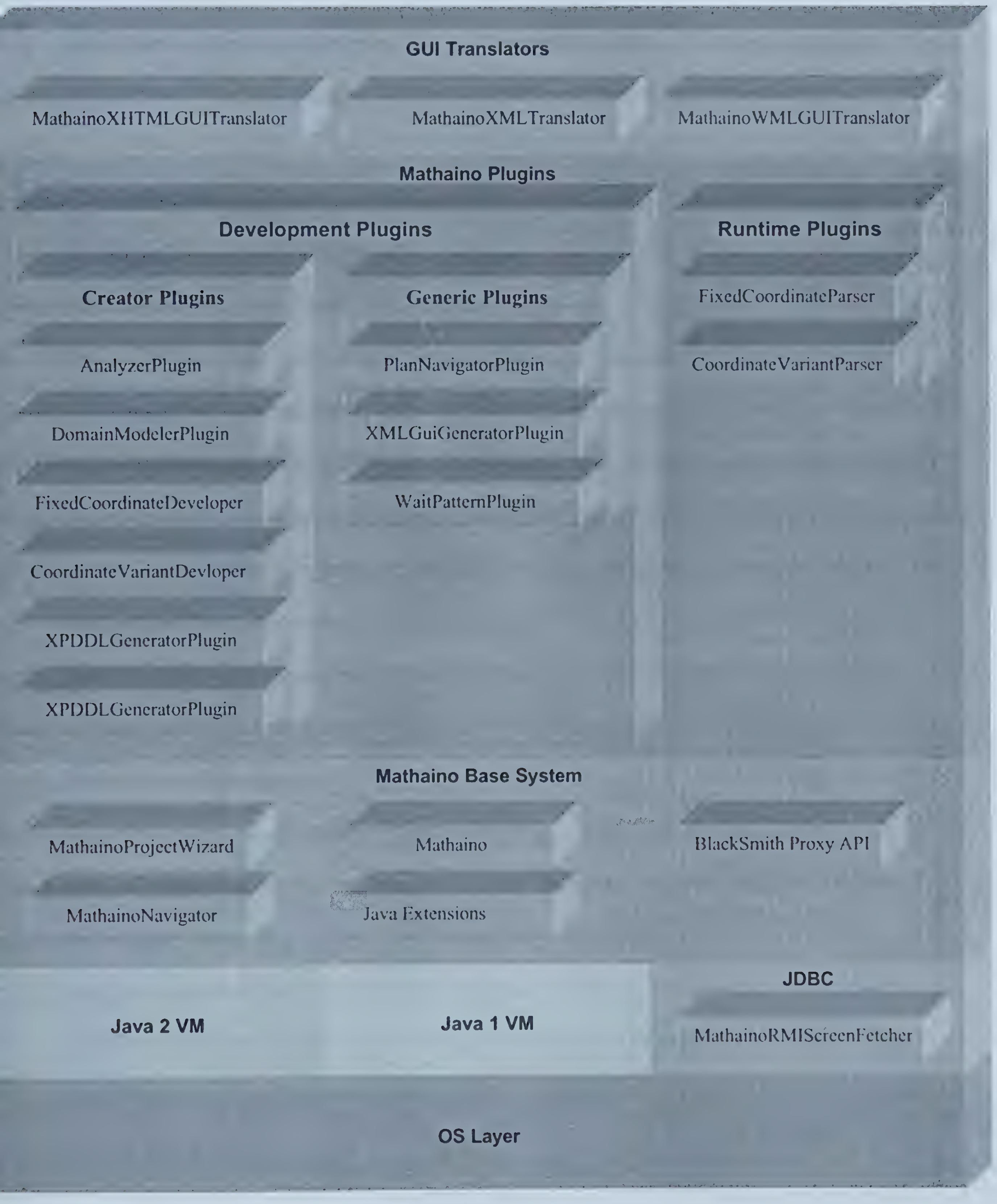


Figure 2.1: Mathaino Architecture

Layer four houses the various components that constitute Mathaino's plugin framework. The various Mathaino plugins can be classified into the following two broad categories:

1. Mathaino Design Time Subsystems
2. Mathaino Run Time Subsystems

The design time sub-components of Mathaino are integrated into a single design and development environment that is used by the developer to analyze the information flow of the legacy system. The main aim of this analysis is to identify the minimum set of input information that is required from the user for accomplishing a particular task on that system. Thus, the aim is to compress the set of actual input information required from the user, leading to development of an optimum GUI for the given legacy system. The data produced by design time subsystems is then utilized by the run time subsystems to access the legacy system on various new platforms. The platform specific runtime components constitute the highest layer in the Mathaino architecture.

It is this division of the various components into two distinct categories, at an architectural level, that allows Mathaino to port the legacy systems in a platform independent manner. While the run time subsystems of Mathaino are migration platform specific, as they try to re-implement the legacy system on a specific target platform, the design time components are completely platform independent.

2.2 Architecture of A Mathaino LIS Wrapper Application

Mathaino is an automated migration tool for simultaneous migrating legacy information systems to multiple platforms. Thus, each Mathaino generated LIS wrapper application has a specific architecture too. In this section, we discuss the general architectural pattern followed by any LIS wrapper application that has been generated using Mathaino.

Usually the three tier architectural pattern is the preferred architectural pattern employed by a majority of black box migration solutions. The three-tier pattern is particularly well suited to these systems as the primary task of any black box solution is to operate as a translator between the thin target client and the backend legacy information system. Since, this single step translation technique mandates that the migratory application reside in between the client and the legacy system the three-tier architectural pattern is a good fit. Also one of the advantages of the three-tier architecture is reusability [SADOSKI00], which is of prime importance when it comes to migration solutions. When compared with the traditional three-tier architectural pattern, as discussed in [SADOSKI00], the black box migratory application sits at the business or process management layer, while the wrapped application forms the presentation layer. The legacy system forms the data access layer. Figure 2.2 illustrates this generic architectural pattern.

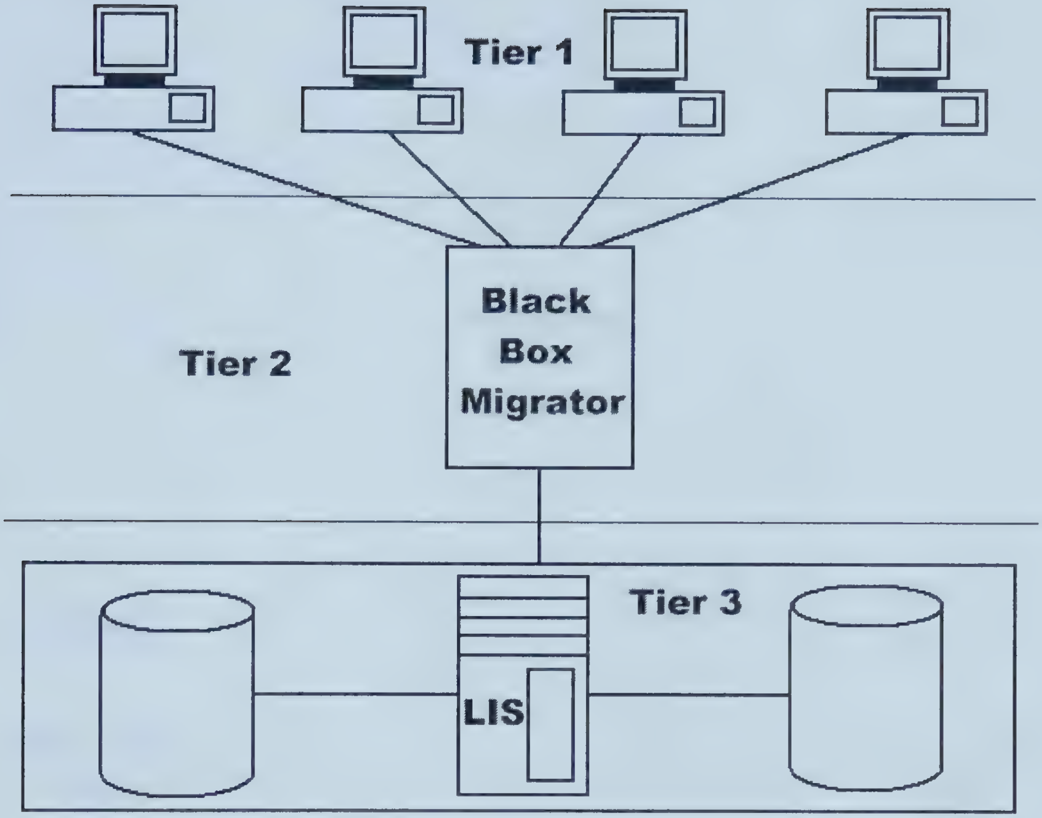


Figure 2.2: Three-Tier Black Box Migration Architecture

A very common example where this architecture is employed the screen scrapping solution for black box migration of legacy information systems. In screen scrapping, the middle tier consists of the TCP-3270 to HTML form translation subsystem.

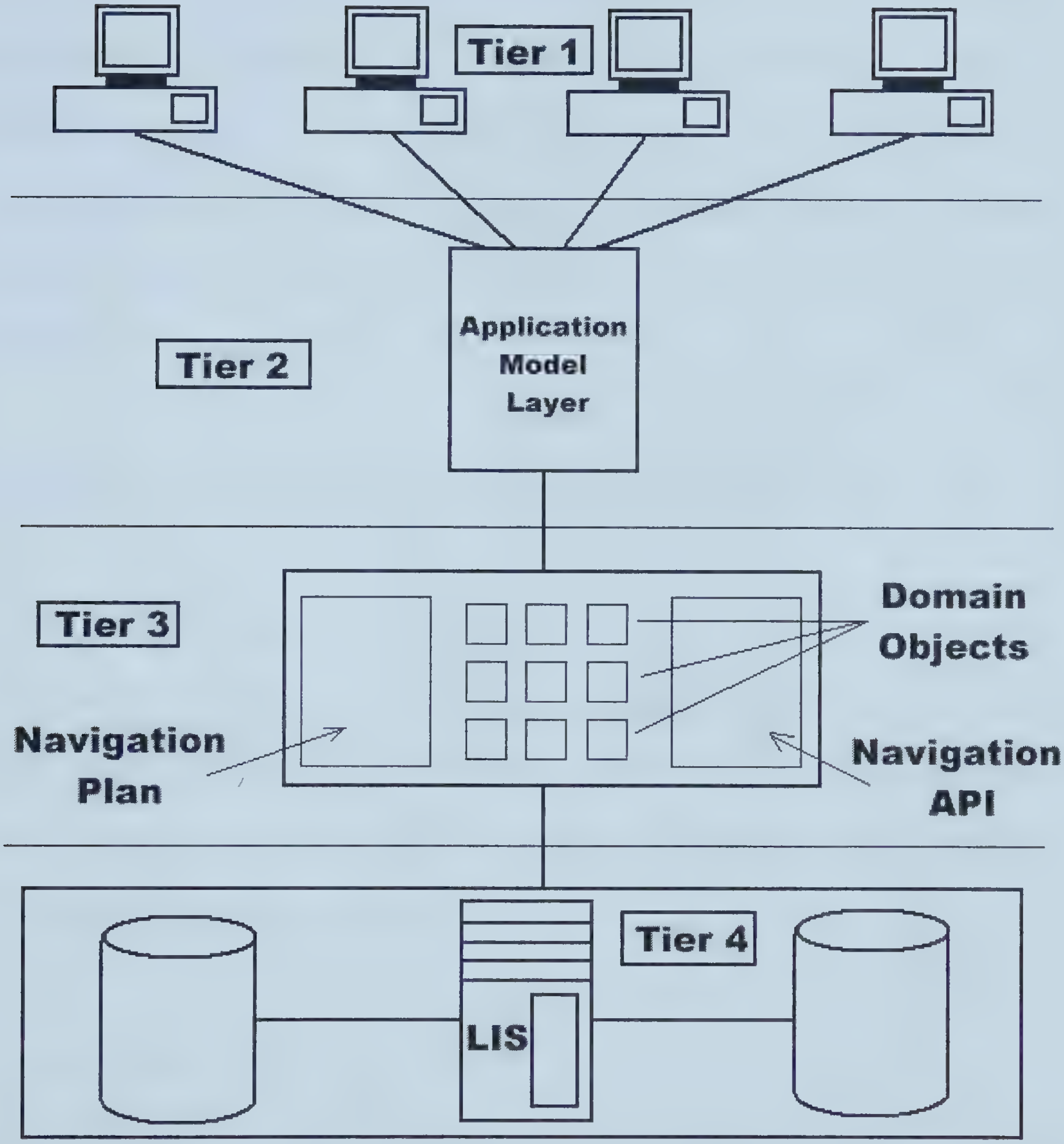


Figure 2.3: Mathaino LIS Wrapper Architecture

While, the three-tier architectural pattern is a good fit for a majority of black box migration solutions where the primary aim is to efficiently migrate a given legacy information system to a particular platform, this scheme is not an effective solution in case of Mathaino. The primary motivating factor for digressing from this architectural

design pattern for us was the requirement that Mathaino should be able to migrate a given legacy information system simultaneously to multiple target client platforms. Since, the three-tier design pattern was not suitable for this task, we decided add another tier making Mathaino generated LIS wrapper applications conform instead to a four-tiered architectural pattern. The generic architecture of a Mathaino generated LIS wrapper application is shown in Figure 2.3.

The various tiers of this architecture are explained below [FOURLAYER]:

1. **The View Layer or Tier 1**

   The view layer fulfills the same function as the presentation layer in the three-tiered architecture. That is, it houses the platform specific user interface built around the native widget set for that platform. This layer is generated completely automatically by the Mathaino development IDE.

2. **The Application Model Layer or Tier 2**

   This layer mediates between the various user interface components on the view layer and the platform independent Mathaino runtime, which resides in the next tier. In case of Mathaino, this layer is also generated automatically by the development IDE.

3. **The Domain Model Layer or Tier 3**

   This is the tier at which the platform independent Mathaino runtime resides. Messages passed on to this layer are completely platform independent and conform to

the abstract model required by the Mathaino runtime. The various domain objects for the LIS being migrated are housed at this layer along with the LIS navigation plan and the navigation API.

4. **The Infrastructure Layer or Tier 4**

   Classically this is the layer where the objects or systems that represent the connections to entities outside the wrapped application reside [FOURLAYER]. In case of Mathaino, this layer contains the wrapped legacy system. We use the BlackSmith terminal emulation API to initiate and sustain connections between this tier and the Mathaino runtime at tier three.

As pointed out previously, the best architectural solution is the one that is dictated by the specific requirements of the particular application [BECK94]. For Mathaino generated LIS wrappers the primary requirement is platform independent operation. Since, the application layer (or tier two) allows us to translate platform specific user interface messages into the platform independent form understood by the backend domain model layer this architectural solution was best suited for Mathaino.

Thus, once the Mathaino development IDE has been used to create and appropriate domain model for a given legacy information system, this four-tier architectural pattern allows us to migrate any given legacy system to a wide range of target client platforms. Further, since the *"main brains"* of Mathaino reside in the fat domain model layer, this

architecture also allows us to quickly develop and deploy relatively thin platform specific translators for a given target platform.

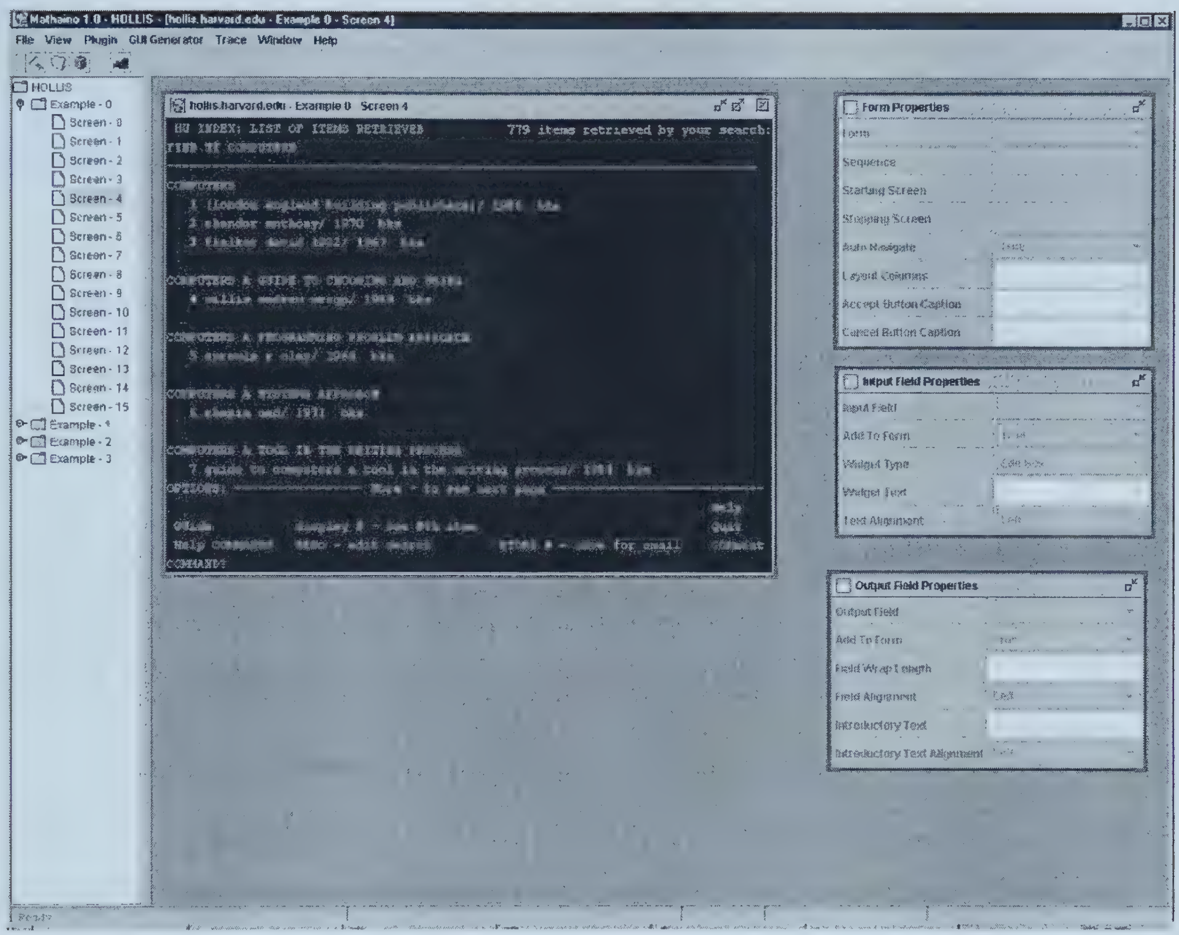
The use of this four-tier architectural pattern for LIS wrappers is unique to Mathaino and helps us in offering an additional benefits like simultaneous multiple platform migration over competing black box migration technologies like batch processing and screen scrapping.

Chapter 3

The Mathaino Developer

Mathaino adopts a two-stage process for migrating a given legacy system to a range of new platforms. Also as shown in Figure 2.1 (please refer to Chapter 2), the entire range of Mathaino plugins can be divided into two categories: the development plugins and the runtime plugins. Thus, corresponding to this two-stage migration process every Mathaino module is either a part of the design and developer subsystem or the runtime subsystem. Although it is not obvious from the architectural diagram, the components in the base Mathaino subsystem which forms the second layer the architectural diagram, are also a part of the Mathaino developer subsystem.

The design time sub-components of Mathaino are integrated into a single design and development environment that is used by the developer to analyze a given legacy information system. Figure 3.1 shows a screen snapshot of Mathaino's IDE.



```
HU INDEX; LIST OF ITEMS RETRIEVED          779 items retrieved by your search:
[illegible]
--------------------------------------------------------------------------
COMPUTERS
  1 [illegible]
  2 [illegible]
  3 [illegible]

COMPUTERS A GUIDE TO [illegible]
  4 [illegible]

COMPUTERS A [illegible]
  5 [illegible]

COMPUTERS A [illegible]
  6 [illegible]

COMPUTERS A TOOL IN THE WRITING PROCESS
  7 guide to computers a tool in the writing process/ 1981 bks
OPTIONS: ---------------- More - to see next page ----------------
                                                                Help
[illegible]                                                     Quit
[illegible]                                 [illegible] for email   Comment
COMMAND?
```

Figure 3.1: The Mathaino IDE

In this chapter we will explain the various important subsystems that comprise the Mathaino developer and explain them in detail including the algorithms employed by them. We will cover the Mathaino runtime in Chapter 4. To explain the algorithms better we will be discussing our simultaneous migration of the Pine email reader to XHTML, WML and XML platforms using the Mathaino developer and runtime as a running example for these two chapters.

The first step in migrating a given legacy system is known as the development phase. During this phase a qualified Mathaino development engineer utilizes the various

Mathaino design and development subsystems that are accessible through the Mathaino IDE to construct an abstract model of the legacy system being migrated. Broadly, the development of this abstract model comprises of three separate stages:

1. The Recording Phase
2. The Analysis Phase
3. The Final System Design Phase

Each of these phases along with the sub-components that participate in them are discussed in the following sections.

3.1 The Recording Phase

Mathaino is a **task based** black box migration solution for legacy systems. Hence, tasks form the core of the migration process as Mathaino has the ability to learn the semantics of performing a given task on a given legacy system. As pointed out in Chapter 1, at the core of Mathaino's developer lie intelligent learning based algorithms that require a sufficient number of training examples in order to learn the process of carrying out a given task on the given legacy system. The recording phase of the development process is required explicitly to fulfill this need for training examples and has to be carried out prior to any other migration activity. Thus, this recording phase represents the first stage of the Mathaino migration process.

Mathaino primarily uses an intelligent and flexible terminal emulator, henceforth referred to as the Lendi Recorder, to execute the recording phase. This recorder subsystem for Mathaino has been largely developed around the existing Lendi Recorder that has been developed as part of the Lendi project. The Lendi project is another project under the umbrella of the Cellest group of projects of which Mathaino too is a part [STROULIA00].

As highlighted in the previous section, Mathaino is a task based migration solution. Hence, the recording process for a given legacy system is task specific too, i.e. it requires an expert user for that legacy system to carry out a specific pre-determined task on that legacy system. We have made an effort to make the recording process as unobtrusive and user friendly as possible. Usually the Lendi Recorder replaces the normal terminal emulation program used by the expert user to log onto the legacy system. Thus, the expert user logs onto the given legacy system using the Lendi Recorder and carries out the specified task just as he would have on this original terminal emulator program. However, as the expert user carries out this task, the Lendi Recorder keeps track of the screens navigated by the expert user and the set of input provided by him to the legacy system for the purpose of carrying out this task.

A single such recording sequence is referred to as a task trace. Based on our experience it seems that Mathaino requires anywhere from two to eight such traces of the given task to serve as its training examples, depending upon the complexity of the given task. We have

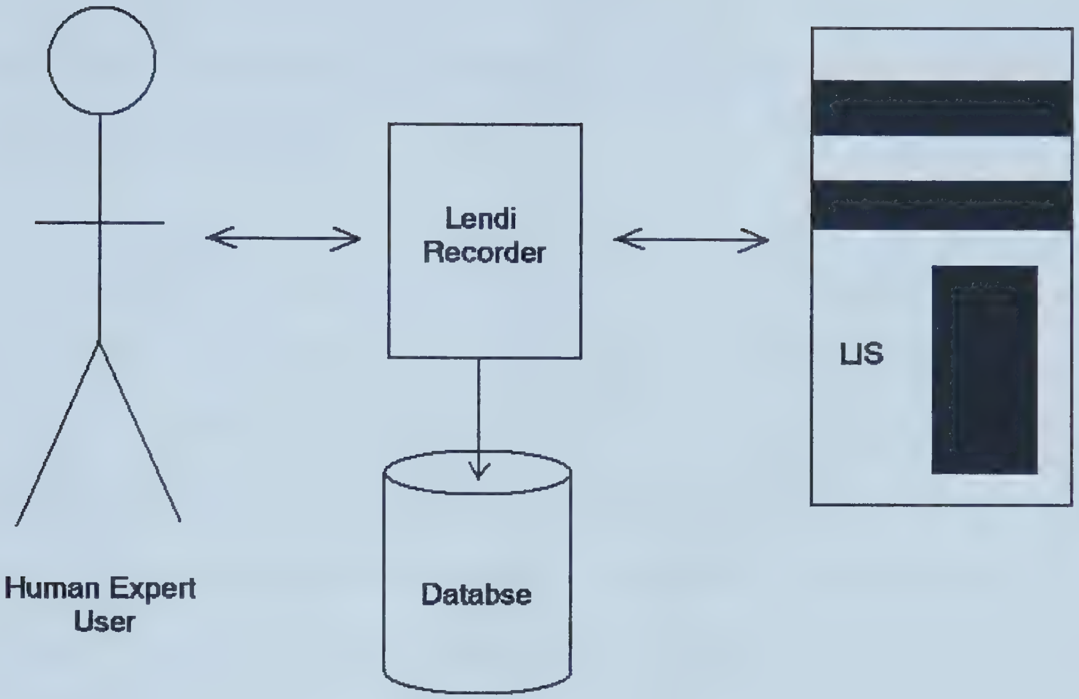


Figure 3.2: Lendi Recorder Setup

observed that in most cases an increase in the number of the training examples causes a corresponding increase in accuracy of the subsequent phases of the development process.

Figure 3.2 illustrates the setup required for recording task traces given a legacy system.

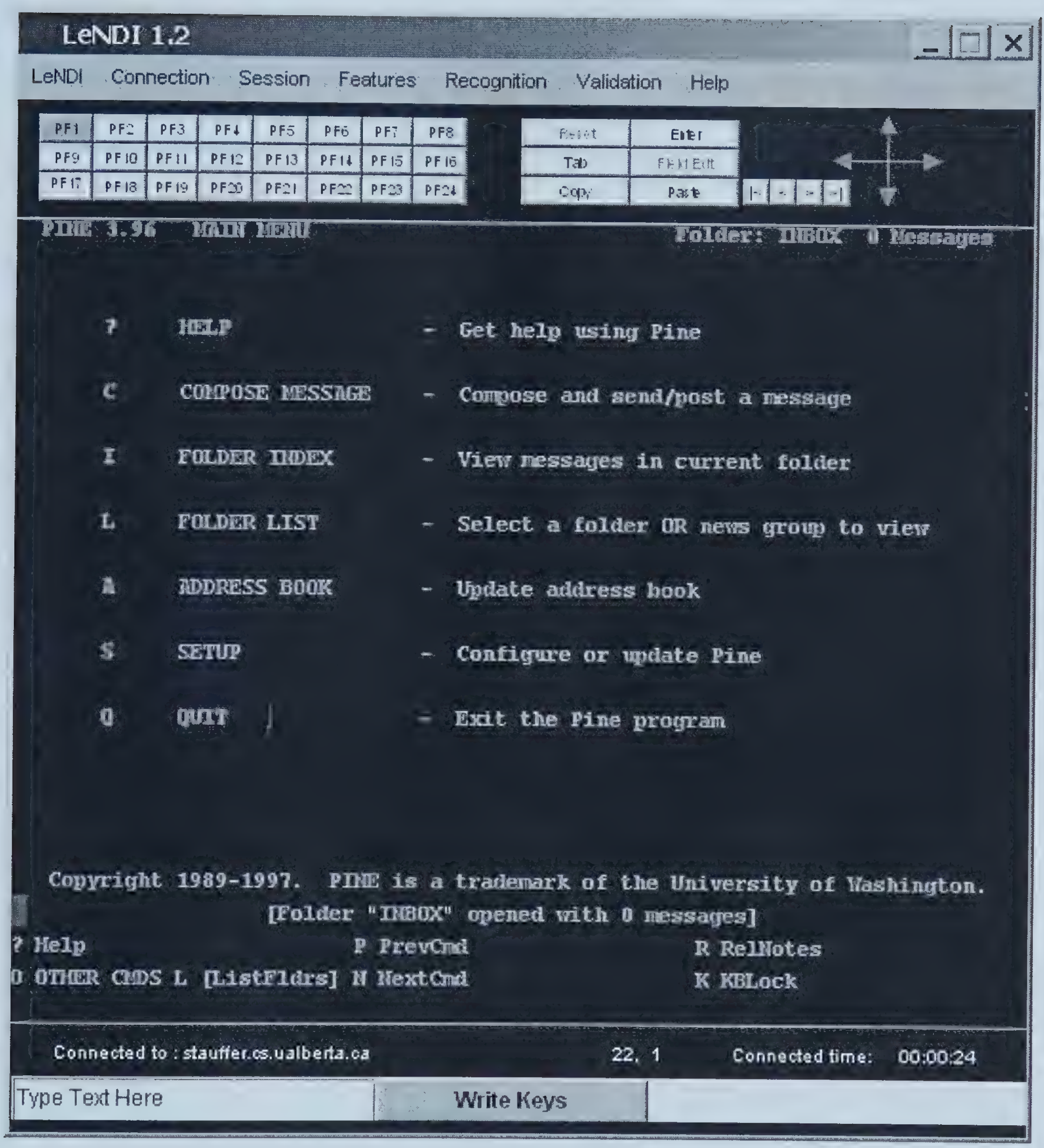
As illustrated in Figure 3.2, the process of recording these task traces is completely transparent to the expert user, for whom for all intents and purposes the Lendi Recorder behaves similarly to the original terminal emulator program used by him.

When compared with the existing legacy system migration solutions processes, this recording phase is equivalent to an expert user interview conducted by a human systems

analyst. The advantages of using the Lendi Recorder over the manual interview process are automation, increased accuracy, cost efficiency, decreased waste of expert user time and, of course, user friendliness.

Once the Lendi Recorder has been used to execute a task trace it stores the entire trace contents inside a Microsoft Access database. During a recording session, the Lendi Recorder records artifacts like the sequence of screens traversed for execution of a particular task on the given legacy system, along with the entire contents of those legacy screens, as well as the entire set of input actions performed by the expert user on each and every screen encountered by him during the enactment of the given task. Since we rely on the recorder being as exhaustive as possible while recording a task trace, it records a very rich set of information about each action that the expert user performs to cause a state transition in the legacy system. We use the term *Action Item* for a single such set of values. The artifacts associated with each action item are:

1. The raw sequence of keystrokes input by the expert user.
2. The result of that action item, i.e. whether it caused a screen or a field transition. This is important as each action item might not navigate the legacy system to a new screen state (which is a visible state transition) but it still does cause a state transition for the given legacy system to a new state.
3. The on-screen location for that action, i.e. the screen coordinates at which it was performed.



```
PINE 3.96   MAIN MENU                          Folder: INBOX  0 Messages


    ?     HELP               -  Get help using Pine

    C     COMPOSE MESSAGE    -  Compose and send/post a message

    I     FOLDER INDEX       -  View messages in current folder

    L     FOLDER LIST        -  Select a folder OR news group to view

    A     ADDRESS BOOK       -  Update address book

    S     SETUP              -  Configure or update Pine

    Q     QUIT               -  Exit the Pine program



 Copyright 1989-1997.  PINE is a trademark of the University of Washington.
                   [Folder "INBOX" opened with 0 messages]
? Help                    P PrevCmd                  R RelNotes
O OTHER CMDS L [ListFldrs] N NextCmd                 K KBLock

  Connected to : stauffer.cs.ualberta.ca          22, 1     Connected time:   00:00:24
```

Figure 3.3: Using the Lendi Recorder on Pine

4. The syntactic grammer for this action item. This syntactic grammar is derived by looking at the pattern of variation of input keystrokes for this field on a system wide basis and generalizing them into a fixed syntactic notation.

Figure 3.3 illustrates the Lendi Recorder module being used for recording a task trace using the Pine email reader housed on a Sun Solaris server belonging to the Department of Computing Science, University of Alberta.

Besides recording the sequence of screens and the actions performed by the expert user using the Lendi Recorder, the recording phase also entails the process of identifying the significant pieces of output information produced during the execution of the given task. As it is not possible to automatically distinguish between significant output and redundant noise present on a legacy screen, the Mathaino developer is forced to rely on the expert user to identify the important output artifacts produced during the execution of the trace for it. For this purpose we have developed a family of two plugins whose sole task is identification and parsing of the significant output information from given legacy system screens. Each one of these family consists of a pair of cooperating plugins one of which is used at the development time for generating a parsing template for significant output field and the other is used at runtime for actually extracting the required output by parsing the incoming legacy screens. The two plugin families developed by us are:

1. The Coordinate Invariant Plugin Family
2. The Coordinate Variant Plugin Family

The coordinate invariant family of plugins can be used for identifying output fields with constant location coordinates. To identify such fields the user has to simply highlight them on the terminal emulator screen of Mathaino. Once he has done so, Mathaino

memorizes these coordinates and can extract the requisite output information the next time same screen is encountered at runtime.

The coordinate variant family of plugins is used for identifying output fields whose location coordinates may vary in separate instances of the same legacy system screen. This template plugin uses a modified form of the case based reasoning (CBR) algorithm, popularized by the Eliza program [WEIZ83], for extracting the requisite output information.

Teaching Mathaino to extract output information using this family of plugins requires a two stage learning process. In the first stage, the Mathaino development engineer has to identify the starting landmarks for a given coordinate variant output field. Since the field must be uniquely identified, Mathaino assumes that its bounded by a unique starting landmark, which if not found indicates the absence of this particular field of output information. In the next stage, the Mathaino developer is required to identify the terminating landmark that encapsulates the given coordinate variant output field on the screen. However, unlike the starting landmark, the terminating landmark is not required to be unique. Thus, if the terminating landmark is not unique, then the user has to identify the set of all possible terminating landmarks for this field. Once this has been done, Mathaino uses this set of starting and terminating landmarks to extract the requisite output fields at runtime. If the given output field template results in multiple matches at runtime (which can happen if the set of terminating landmarks has more than one member), Mathaino returns the intersection of all the parsed value strings as the required

value for that output field. This is a variation from the classical Eliza algorithm where in case of multiple matches a weighted template scheme is utilized for deriving the final result.

Mathaino relies heavily on this recorded task trace, consisting of the legacy system screens for the enactment of the task, the set of action items for these screens and the set of identified output fields, for the carrying out next phase of the development process known as the analysis phase.

The next section discusses the analysis phase of the Mathaino development process.

3.2 The Analysis Phase

The analysis phase forms the core of the Mathaino development process and is perhaps the single most important stage in the entire migration process employed by Mathaino. The analysis phase itself consists of a number of important sub-phases. The listing below enumerates these sub-phases:

1. Analysis of the set of recorded action items in order to construct a repository of input fields.
2. Classification of the various input fields into subcategories.
3. Generation of the screen recognition database.
4. Generation of non-deterministic navigation plans.

5. Object modeling.

Each of these sub-phases is explained in more detail along with their associated developer plugins in the following sections.

3.2.1 Construction of the Input Field Repository

Any Mathaino migration project consists of two core repositories upon which the rest of migration data structures, including domain objects, are built. One of these contains meta-data about the output produced during the execution of the task trace being migrated. This repository is known as the output field repository for obvious reasons. This repository is constructed automatically and transparently during the output field identification phase, which is the part of the recording phase. However, unlike the output field repository, the input field repository is generated during the analysis phase. This repository contains meta-data about the set of input action items generated by the expert user during the enactment of the task trace. For example, one of the artifacts of this meta-data is the syntactic grammar of each input field deduced on a per-task basis. This is different from the syntactic grammar generated by the Lendi Recorder as that one is generated on a system wide basis and is, thus, inappropriate for Mathaino. The following list enumerates the elements of the set of meta-data generated for each action item in the input field repository:

1. The parsed termination sequence (dispatch command) for that action item. This is essentially a translation from the raw keystrokes recorded by the Lendi Recorder.

2. The actual parsed data content for that input field. This is essentially the sequence of parsed raw keystrokes minus the terminating sequence. While parsing the raw keystrokes the Mathaino analyzer makes sure that it translates certain special BlackSmith character sequences into their appropriate ASCII notation.

3. The task based syntactic grammar for the input field. This is constructed by analyzing the variation in the parsed data section of the input field on a per-task basis. For example, if for a specific action item the user always enters a random sequence of keystrokes followed by a fixed sequence then its corresponding syntactic grammar would be `[*]Fixed Sequence`. An example of such an action item is the read email command in Pine whose syntactic grammar is `ij[Email Number To Read]`. Parsing out this grammar is useful as it helps in shortening the amount of actual input information required from the user for such an input field. For example, once we figure out the syntactic grammar for the read command for Pine, all that we require from the user at runtime is the actual number of the email to read, since we can deduce that `ij` has to be appended to it from the syntactic grammar.

3.2.2 Categorizing the Input Fields

The input field analyzer plugin is responsible for categorizing the various input fields that constitute the input field repository discussed in the previous section. This plugin is one of the core components of the Mathaino IDE. It is responsible for analyzing and classifying the input fields that comprise the net information flow to and from the legacy system.

The main purpose behind analyzing the flow of information to and from the legacy system is to identify the minimum set of input information that is required from the user for accomplishing a particular task on that system leading to development of an optimum GUI for the given legacy system. During this categorization phase the Mathaino input field analyzer tries to classify each input field into one of the following sets:

1. **Trace Constant:** This is an input field, whose value is the same in all example task traces.

2. **Derived:** An input field, whose value is obtained through a system-display interaction and is subsequently provided as input to the system through a user-entry interaction.

3. **Redundant:** An input field, whose value is provided as input to the system through multiple user-entry interactions.

4. **Range:** An input field, whose value varies within a well-defined range for all example traces.

5. **Unpredictable:** An independent user-input information entity.

Mathaino's input field classification subsystem categorizes each data element into one of these categories by simultaneously evaluating the value of the data item in the various example traces.

First, if a data item, manipulated at a particular point in the trace, has the same value for each example, the input field analyzer infers that it is a constant entity, and memorizes the particular value for the entity. These entities are manipulated automatically by the run-time environment, while executing instances of the same task. Usually a majority of navigation commands for a legacy system can be grouped under this category, as usually they tend to be constant over all the example traces.

In the next phase, the input field analyzer examines the data items to identify those that are input by the user and have the same value as other data items previously displayed by the system. All such input fields are classified as derived. The run-time environment requires the entry of the entity value once and automatically enters this value on all subsequent states where it is required. Such entities tend to be identification keys that are used to retrieve different pieces of related information from different databases. Similar to derived entities, redundant ones are input by the user at some state and have the same value as another entity input in some previous state.

As Mathaino analyzes the given example, it keeps track of the set of unique values input for any particular input field. In cases where, over a large number of examples, the input field analyzer detects only a much smaller variation of the set of input values for that

field, it identifies that input field as a range entity and memorizes the set of possible values for it. This is used in the design of the new interface, where this entity can be input using a selection widget as opposed to an entry widget.

The input fields that can't be categorized into any of these previous sets are marked as unpredictable. This set constitutes the minimum set of user input that is actually required for performing the given task on the legacy system. Usually this categorization of input fields leads to a significant reduction in the actual input information required from the user.

To illustrate this process better let us consider the task of reading a specific email using the Pine email reader. To execute this task the expert user has to issue the following commands in the given sequence:

1. Enter the login name
2. Enter the password
3. Issue the UNIX command "`set term=vt100`"
4. Issue the UNIX command "`pine`"
5. Open the Pine inbox using the command "`i`"
6. Issue the Pine jump to message command "`j`"
7. Type the number of the email to read
8. Issues the Pine quit command "`q`"
9. Issue the UNIX logoff command "`exit`"

Sequence Number	**Field Content**	**Categorization**
1	Login name	Unpredictable (Required)
2	Password	Unpredictable (Required)
3	`set term=vt100`	Constant (Not Required)
4	`pine`	Constant (Not Required)
5	`i`	Constant (Not Required)
6	`j`	Constant (Not Required)
7	Mail Number	Unpredictable (Required)
8	`q`	Constant (Not Required)
9	`exit`	Constant (Not Required)

Table 3.1: Categorization of Commands from the Pine Trace

As is obvious, most of these commands are simple navigation commands that remain constant over traces. Table 3.1 lists the categorization of these commands deduced by the input field analyzer plugin using the previously discussed algorithms.

As illustrated in Table 3.1, the actual set of input information required from the user gets compressed to just three commands from the previously listed nine commands after the input field analyzer has finished analyzing the recorded action items. This evaluates to about 66% reduction in the amount of input information required from the user for enacting this particular task trace!

3.2.3 Generation of the Screen Recognition Database

The majority of legacy systems are organized around a central mainframe computer that is accessed by simple text terminals. Thus, in order for the wrapper application constructed by Mathaino to replace the human user, it needs, not only an effective terminal emulation software API, but also an effective schema for automated screen recognition, so that it can always infer the current execution state of the backend legacy system. The knowledge of the current system state is necessary to verify whether the current navigation plan is progressing as expected and also to make necessary plan adjustments as and when they become necessary. Thus, the screen recognizer plugin plays a pivotal role in the entire development process.

Mathaino learns a predicate for recognizing the distinct screens, through which the user navigates, using a text-document clustering method. Mathaino assumes that its input example traces are parallel, i.e., that all the examples of the system user interaction are instances of the same high-level plan and therefore contain the same set of visited screens in the same order. Thus, it clusters the corresponding screen instances of the different example traces and learns a "keyword signature" for each of them. Later on, at run-time, these signatures are utilized to classify the new screen instances of the legacy user interface into the original clusters, and, thus, recognize the new screen instances and the corresponding behavioral states of the system.

Document clustering algorithms can be classified in two broad categories:

1. Example based clustering systems

2. Automated clustering systems

While classical example-based clustering systems use a 100% accurate pre-clustered small document collection for clustering a larger set of documents [AGGRAWA99], automated clustering systems rely on the expert user to modify the cluster models that have been developed by them automatically. Mathaino's algorithm is a modification of the example-based clustering algorithms that have been proposed previously (see [AGGRAWA99]).

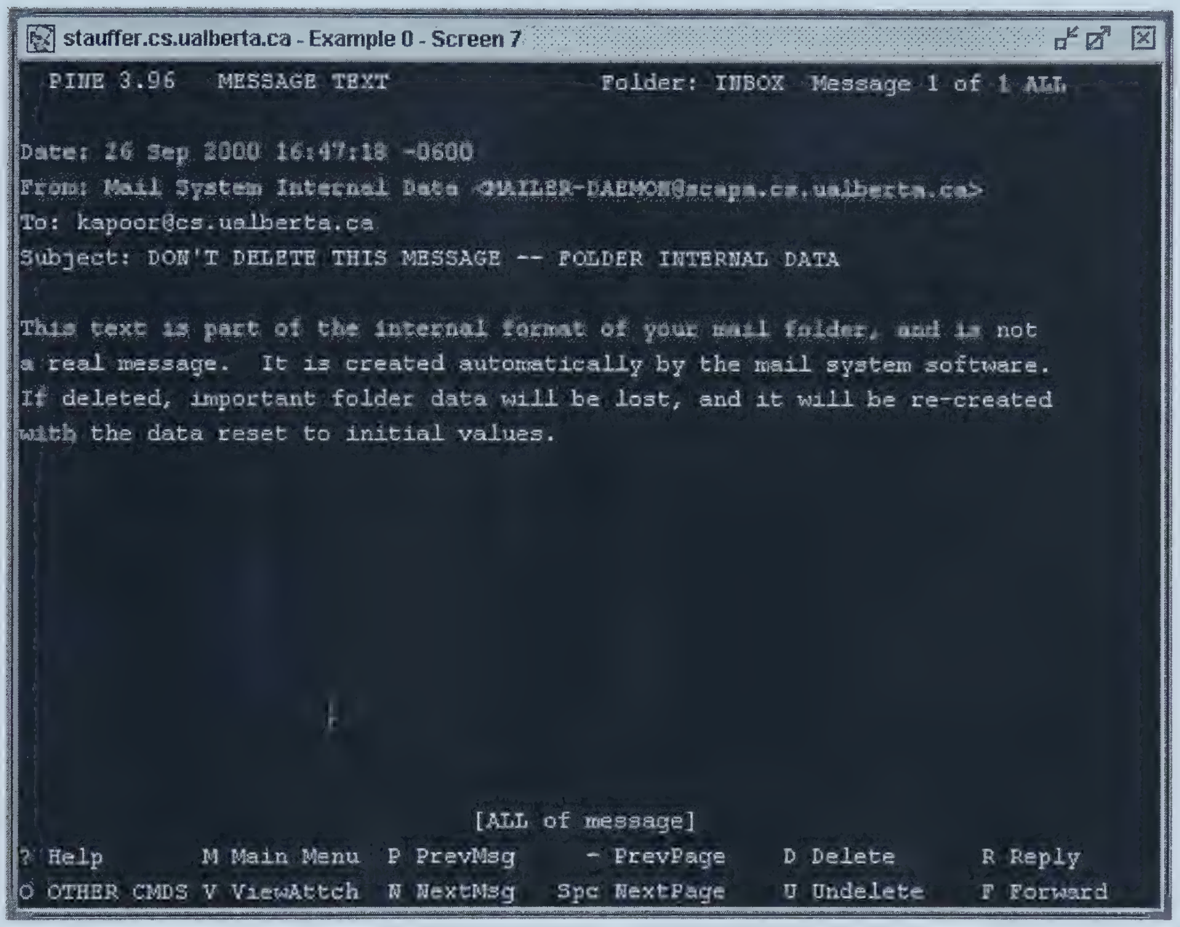
In the context of the Cellest project, we have developed an automated screen clustering system, called Lendi [STROULIA99] that can interactively cluster legacy screen snapshots, with an estimated accuracy of 90%. Lendi uses “random”, as opposed to task-specific traces, as input, and using a set of features designed specifically for the legacy interface domain, uses a single-pass, centroid clustering algorithm to cluster all the screen snapshots included in the traces in same-screen clusters. Lendi’s clustering is effective, however, Mathaino’s runtime requires 100% clustering and classification accuracy, otherwise it will not be able to recognize the legacy screens at run-time and will get “lost”.

To improve upon Lendi’s clustering abilities, the screen recognizer plugin starts by assuming that Lendi has accurately grouped a majority of the screens in the given

clusters. The next step is to build the signature for each cluster by identifying the set of keywords that occur in at least ρ screens in that cluster where ρ is the assumed confidence level of the previous clustering algorithm and $\rho > 0.5$. This set of keywords forms the probable signature for this cluster. The next step of the algorithm is to identify the maximum set of keywords, within this previously identified set, that are simultaneously contained in at least a fraction of screens that is greater than or equal to ρ in the of the given cluster. In essence this problem is akin to the problem of exact set coverage, which is a NP-complete problem. However, for Mathaino ρ is equal to 1.0 as the recorded example traces are parallel and, thus, the clusters are well formed. Thus, in this case a much simpler polynomial time algorithm can solve this problem. In either case, it is this smaller set that is then recorded as the signature for this cluster.

The algorithm used by the screen recognition plugin is flexible enough to identify a keyword even if it migrates within a fixed column or row range on the legacy system screen.

Since this algorithm is based on keyword recognition it is possible for the user to modify the internal signature by simply correcting the document collection in a cluster if it is malformed. In case the user decides to perform this step, the screen recognizer plugin recalculates the set of keywords that constitute the signature for that cluster by including or purging the keywords that initially caused the documents to be clustered incorrectly. In this way the screen recognizer plugin can automatically modify its internal cluster model by using user feedback.



```
  PINE 3.96   MESSAGE TEXT                Folder: INBOX  Message 1 of 1 ALL

Date: 26 Sep 2000 16:47:18 -0600
From: Mail System Internal Data <MAILER-DAEMON@scapa.cs.ualberta.ca>
To: kapoor@cs.ualberta.ca
Subject: DON'T DELETE THIS MESSAGE -- FOLDER INTERNAL DATA

This text is part of the internal format of your mail folder, and is not
a real message.  It is created automatically by the mail system software.
If deleted, important folder data will be lost, and it will be re-created
with the data reset to initial values.












                              [ALL of message]
? Help       M Main Menu  P PrevMsg      - PrevPage    D Delete      R Reply
O OTHER CMDS V ViewAttch  N NextMsg    Spc NextPage    U Undelete    F Forward
```

Figure 3.4: An Image of a Pine User Interface Screen

At run-time, as each screen is received by the Mathaino run-time subsystem, it is matched against the cluster signature for the screen that Mathaino expects to be at according to the plan it is executing. In case this screen fails to match the required signature, Mathaino tries to account for it by switching to an alternate plan for carrying out the same task. This scheme is discussed in more details in the next section.

Sequence Number	**Keyword**	**Coordinates [X,Y]**
1	PINE	[3, 1]
2	MESSAGE	[15, 1]
3	TEXT	[23, 1]
4	Folder	[42, 1]
5	INBOX	[50, 1]
6	Help	[3, 23]
7	PrevMsg	[29, 23]
8	Reply	[71, 23]
9	NextMsg	[29, 24]

Table 3.2: Identified Keywords for the Screen shown in Figure 3.4

Figure 3.4 illustrates an example screen from the Pine email reader program. Some of the keywords, identified using the previously discussed algorithm, that constitute the unique the runtime signature for this screen are listed in Table 3.2.

3.2.4 Generation of Non-deterministic Navigation Plans

After the Input Field Analyzer plugin has finished the analysis of the remote system Mathaino creates an internal navigation plan for the remote system. Essentially, this plan details the regions of the navigation plan that Mathaino can traverse automatically, the

pre-conditions and post-conditions of any navigation action, and the effects produced by it.

To enable the seamless integration of its various subsystems, Cellest adopts XML [XML1.0] as the syntax for information exchange among its subsystems. The Mathaino plan generator plugin adopts PDDL [MCDERM98] for the representation of the plans it learns; therefore we have developed an XML language for representing PDDL-like plans for navigating the legacy interface, called XPDDL.

At run-time, Mathaino's PDDL plans guide the run-time environment to execute the necessary sequence of interactions with the underlying legacy system while also interacting with the user through the new interface to receive the required information entities.

A Mathaino navigation plan consists of high-level interaction operations carried through the new interface constructed by Mathaino, and low-level interaction operations executed on the existing legacy interface. Essentially, through the first type of operations the run-time environment receives from the user the values of the unpredictable information, and displays the values of all output information entities, and through the latter type it enters this information to, and extracts it from, the existing system interface.

A particular task may be performed in several different ways, i.e., through different navigations in the legacy interface. To accommodate for this possibility, currently

Mathaino requires the expert user to explicitly annotate the example traces to indicate the screens on which alternative actions leading to alternative destination screens may be performed. Then it proceeds to analyze the corresponding alternative plan branches as described in the previous sections. At run-time, transparent to the user, Mathaino detects plan branching when it occurs, by recognizing which of the alternative destination screens has been reached using the data generated by the screen recognizer plugin.

```
<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE mnpddlml SYSTEM "mnpddlml.dtd">

<mnpddlml>
 <domain>
  <system>stauffer.cs.ualberta.ca</system>
 </domain>
 <problem>
   <name>Read_An_Email</name>
   <plan>
       . . .
    <action>
     <name>NavigateScreen</name>
     <precndt>
       <scr>1</scr>
     </precndt>
     <effect>
       <scr>2</scr>
     </effect>
     <precndt>
       <input>%(Login)</input>
     </precndt>
    </action>
       . . .
    <action>
     <name>NavigateScreen</name>
     <precndt>
       <scr>3</scr>
     </precndt>
     <effect>
       <scr>4</scr>
     </effect>
     <precndt>
       <input>set term=vt100@E</input>
     </precndt>
    </action>
       . . .
    <action>
     <name>NavigateScreen</name>
     <precndt>
       <scr>7</scr>
     </precndt>
     <effect>
      <scr>8</scr>
      <output>%(Message)</output>
      <output>%(EmailDate)</output>
      <output>%(EmailFrom)</output>
      <output>%(EmailSubject)</output>
      <output>%(EmailSub)</output>
     </effect>
     <precndt>
       <input>qy@E</input>
     </precndt>
    </action>
       . . .
   </plan>
  </problem>
 </mnpddlml>
```

Figure 3.5: XPDDL Plan for Reading an Email using Pine

Figure 3.5 lists the navigation plan for the task of reading an email using the Pine email program.

The semantics of the various elements present in the DTD for XPPDL are explained below:

1. **mnpddlml:** Stands for "Mathaino PDDL Markup Language"; indicates the start of an XPDDL document.

2. **mnpddlml::domain:** Indicates the domain for this plan and the legacy system on which this domain exists

3. **mnpddlml::domain::system:** Encodes the URI of the legacy system to be used for the enactment of this plan.

4. **mnpddlml::problem::name:** Indicates the problem being solved by this plan.

5. **mnpddlml::problem::plan:** This element encodes the plan to be used for solving this problem.

6. **mnpddlml::problem::action:** Indicates an action that must be accomplished to solve this problem.

7. **mnpddlml::problem::action::name:** Lists the name of the action to be taken. In the given example the name *NavigateScreen* means that the legacy system should go to the next visible transition state once this action has been completed.

8. **mnpddlml::problem::action::precndt:** Lists the precondition to be satisfied before the execution of the given action.

9. **mnpddlml::problem::action::effect:** Lists the effect of this action.

10. **mnpddlml::problem::action::precndt::input:** Lists the input field that must be sent to the legacy system so that the given action can be performed.

11. **mnpddlml::problem::action::effect::output:** Lists the output field that would be produced by the legacy system in response to this action.

Thus, the XPDDL plan generator is capable of generating a complete navigation plan in a rich XML notation of PDDL for enacting the given task on the given legacy system once the previous analysis sub-phases have been successfully completed.

The Mathaino IDE also includes a Plan Navigator plugin. The plan navigator plugin allows the Mathaino developer to textually verify that the generated plan for the given task is indeed correct before he proceeds to generate a new GUI for the remote system. In case he detects any ambiguities in this plan he can use the QandA system to make

appropriate modifications to rectify the problem. The QandA project is another project under the Cellest umbrella one of whose aims is to construct a visualizer for Mathaino's navigation plans so that user can visually inspect and modify them if needed.

3.2.5 Domain Modeling

The domain modeler subsystem of Mathaino is an object oriented design system that is used to decompose the task domain for the given legacy information system into discrete domain objects.

An example best explains this process. We find that when we read an email using the Pine legacy email reader, the information fields supplied to the LIS and generated from it are scattered across various input and output fields on different screens of the trace. However, a better means of organizing these fields is by grouping them into a unified set of domain objects, which should then be suitable for any Email domain not just Pine. For example, the first screen of the trace contains two important fields i.e. the login and the password. Again the seventh screen contains important output fields like the email date, subject, return address, and text. Now if we are required to export this set of information for another application or if we are required to carry out this task trace then this is the information we need. However, we can not annotate these fields in a meaningful manner unless we amalgamate related fields into unified domain objects. The domain modeler plugin of Mathaino can be used to create such objects.

The domain modeler plugin also contains a task modeler sub-system. The purpose of this sub-system is to export the input and output field data of the legacy system into an XML data file whose schema is based on the domain objects that have been created by the domain modeler.

Thus, the domain modeler component allows us to not only reform the task domain into unified objects, but it also allows us to export the data store of the legacy system into a new schema that reflects the new domain model.

For our example task of reading an email using pine the set of domain objects consists of:

1. **A User Object:** This contains the login and password fields as `User::Login` and `User::Password`.

2. **An Email Object:** This contains various attributes fields like date, return address, subject and text of the email as `Email::Date`, `Email::Text` etc. and a method for reading the email as `Email::ReadCommmand`.

Once these domain objects have been created using the domain modeler plugin, Mathaino no longer regards the given task in terms of screens and input or output fields. Instead the rest of Mathaino's plugins use this object oriented domain model for interacting with the outside world. Some of the obvious benefits of this scheme are:

1. Research on legacy system migration has pointed out that wrapping a legacy system with objects that faithfully represent their real world counterparts in that domain is a better technique for migrating legacy systems [DELUCIA98]. The concept of domain objects in Mathaino's task model accomplishes the same purpose within Mathaino's domains.

2. It imposes a highly customizable, artificial object oriented model on top of the legacy system without modifying its source code.

3. It allows Mathaino to export legacy system information in terms of domain objects which makes it easier for external applications to make sense of the legacy system data.

4. It allows Mathaino generated wrapper applications to integrate and inter-operate with other applications using the generated domain objects.

5. Since we perform domain modeling at the interface level, it imposes no extra risk or cost on the migration process.

3.3 The Final System Design Phase

The final development phase consists of generating a new abstract user interface for the legacy information system being migrated. This step utilizes the entire set of analysis

information that has been generated in the previous development phases. Mathaino's IDE incorporates an Abstract GUI Generator plugin that is used for producing the abstract GUI in this phase.

3.3.1 The Abstract GUI Generator Plugin

As mentioned previously, the Abstract GUI Generator subsystem unifies the work done by all other design time subsystems of Mathaino. It is this sub-component that is responsible for producing the set of abstract GUI forms that would form the new user interface for the legacy system being migrated. Each abstract GUI form produced by the abstract GUI generator consists of a logical set of screens of the legacy system. Thus, while constructing an abstract form, Mathaino is actually generating a navigation plan for these logical screens that would be used at runtime by the platform specific translators for navigating the legacy system. The complete set of forms, thus, forms the new GUI for the legacy system for accomplishing the given task on it.

3.3.2 Grouping Legacy Screens into Abstract GUI Forms

The Abstract GUI Generator is capable of producing a new GUI for the legacy system, from scratch and completely automatically. It uses the information produced by other design subsystems, primarily the analyzer, to accomplish this. As mentioned previously, each abstract form is a logical grouping of legacy system's screens. Thus, the main issue for the Abstract GUI Generator is resolving the legacy system's screens into logical groups. Currently, the Abstract GUI Generator utilizes an algorithm based on the

observed information flow in and out of the legacy system to generate these abstract GUI forms.

To group the screens into logical forms, the Abstract GUI Generator follows the following steps:

1. **Calculating Points for Form Division**

   A trace of a particular task on the legacy system consists of a linear sequence of screens. The first step in constructing abstract forms is to identify the set of starting screens for each form.

   To calculate the these points for form division, the Abstract GUI Generator assumes that the value of each output field that has been identified previously by the user is essential to deduce the set of values of the required (i.e. unpredictable or range) input fields occurring after it.

   Thus, Mathaino starts a new form as soon as it encounters any screen that contains a required input field, preceded by a previously identified output field.

2. **Normalizing the List of Starting Screens**

   After Mathaino has created the list of starting screens for the various forms according to the previous algorithm, the next step is to normalize this list. Some of the factors that must be accounted for are:

a. The first screen of the trace must be the starting screen for the first form. In case it has not already been included into the list it is added now.

b. To cleanly exit the legacy system it is necessary that the last screen of the trace be the terminating screen for the last form. Thus, in case it has not already been included into the list, it is added now.

c. Also Mathaino makes sure that all output fields identified by the user are shown on some form. Thus, if any screens containing output fields have been ignored, Mathaino creates a new form to accommodate them.

3.3.3 Layout of Abstract GUI Forms

All the abstract forms produced by the Abstract GUI Generator consist of separate output and input areas.

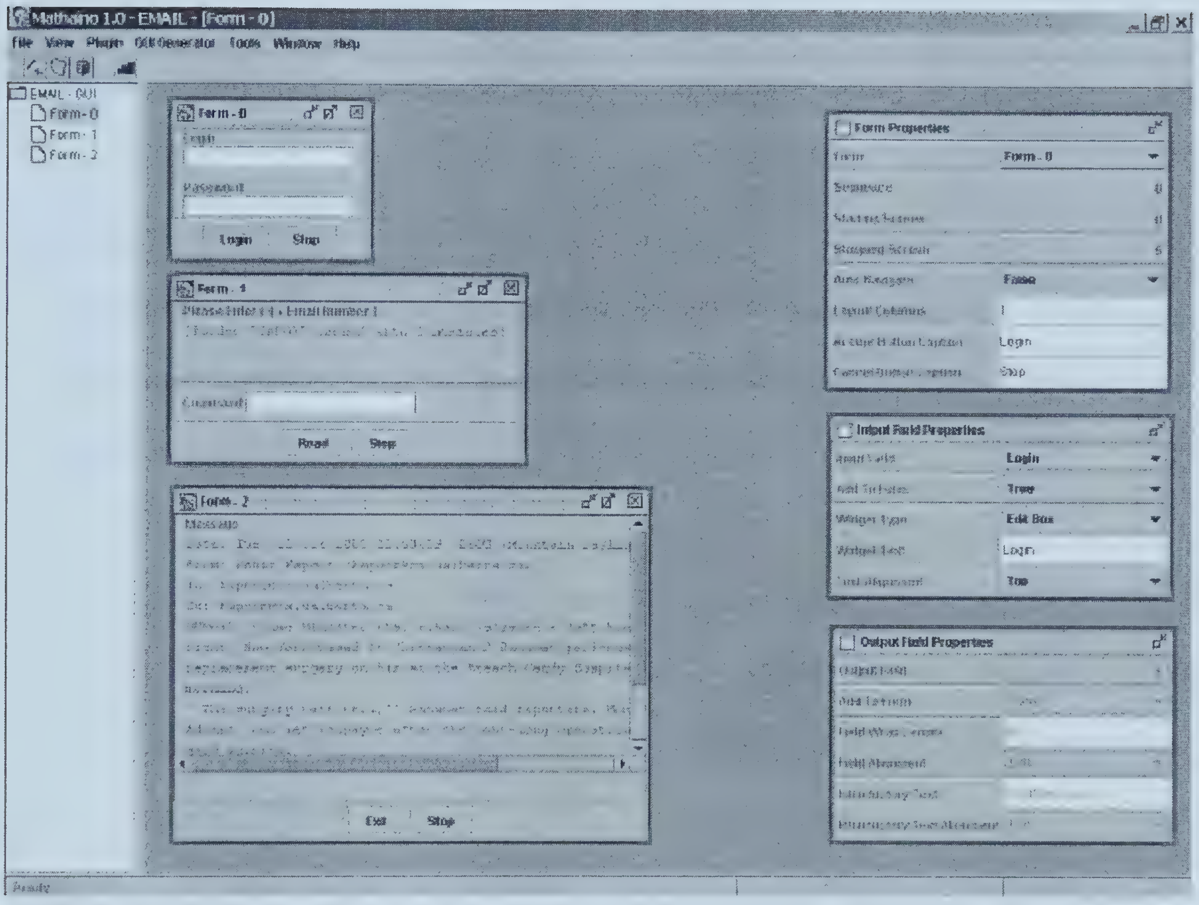
Mathaino lays out each form in a tabular manner. The user is allowed customize this layout pattern by using the visual layout tools of the Mathaino IDE. For example, he can choose to change the number of layout columns as per his requirements. Mathaino automatically rearranges the input widgets in accordance with the new layout columns.

For output fields, the user can change the wrap length for each output field. All the data for that field will then be wrapped at the new wrap length. This is helpful as many text

based legacy systems rely on a particular wrap length to generate a tabular layout of data. However, Mathaino might forsake this setting for devices whose physical screen is not capable of showing the requisite number for characters per line.

The user can also customize the introductory text for both output fields and input widgets. He can choose also various alignments, relative to the input and output fields, for this text.

Previous work on automated user interface generation has already pointed out the fact that automated widget selection decisions must depend upon the type [EISENSTEI00] and characteristics of a given input field [MOORE94]. Mathaino uses a decision tree based algorithm, which is quite similar to the one suggested in [EISENSTEI00], to guess the input widget best suited for a particular input field. Since, the input field analyzer plugin can categorize and also deduce the range of variation of any input field the Abstract GUI Generator plugin uses this information to decide the type and, hence, the GUI widget for an input field. For example, for range input fields Mathaino chooses the combo box if the number of possible choices is more than two and a set of radio buttons otherwise (i.e. for Boolean fields). Similarly unpredictable fields are assigned a simple text box widget by default. However, the user can choose to override these settings and choose another widget for the input field if he so desires. For example, a very common task is to replace the simple input box widget with a password widget for password fields. At runtime, the Mathaino GUI translators try their best to supply a widget for the target platform that closely matches the abstract input widget selected by the user.



Form Properties

Form	Form - 0
Sequence	0
Starting Screen	0
Stopping Screen	5
Auto Navigate	False
Layout Columns	1
Accept Button Caption	Login
Cancel Button Caption	Stop

Input Field Properties

Input Field	Login
Add To Form	True
Widget Type	Edit Box
Widget Text	Login
Text Alignment	Top

Output Field Properties

Output Field	
Add To Form	[illegible]
Field Widget Length	
Field Alignment	[illegible]
Introductory Text	[illegible]
Introductory Text Alignment	[illegible]

Figure 3.5: Generating Abstract GUI Forms for Pine using Mathaino IDE

Figure 3.5 illustrates the set of Abstract GUI forms automatically generated for the Pine email reader, by the Abstract GUI Generator, being rendered inside the Mathaino IDE.

After the user is satisfied with the GUI, Mathaino generates an XML representation for the current set of abstract GUI forms. The GUI translators are responsible for translating this abstract XML GUI, at runtime, into the actual platform specific target GUI. As the basic Mathaino subsystems are platform independent, the translators are also responsible for retranslating the platform specific messages received via the new GUI back into a

platform independent form that can be understood by the backend Mathaino navigation subsystems.

In this manner, the user can create a single, unified GUI representation for his legacy system, which can then be implemented on multiple platforms automatically. The user doesn't have to worry about the specifics of the target platform; the runtime Mathaino translators handle them for him.

Chapter 4

The Mathaino Runtime

The Mathaino runtime components constitute the final stage of the two stage Mathaino migration process first discussed in Chapter 3. The main task of the runtime components is to enact the legacy information system being migrated on a new platform, using the entire set of information that has been produced by the Mathaino developer components in the previous migration phase.

The runtime components constitute the topmost layer of the layered architectural pattern followed by Mathaino (shown in Figure 2.1). Mathaino runtime components are completely reusable and they constitute an important part of any LIS wrapper solution generated by Mathaino. They encompass the application model tier and the domain model tier in the four-tier architecture of a Mathaino LIS wrapper shown in Figure 2.3.

Architecturally, the runtime can be divided into the two distinct sub-layers. The platform specific GUI translation components form the highest layer that sits on top of the platform independent base runtime layer.

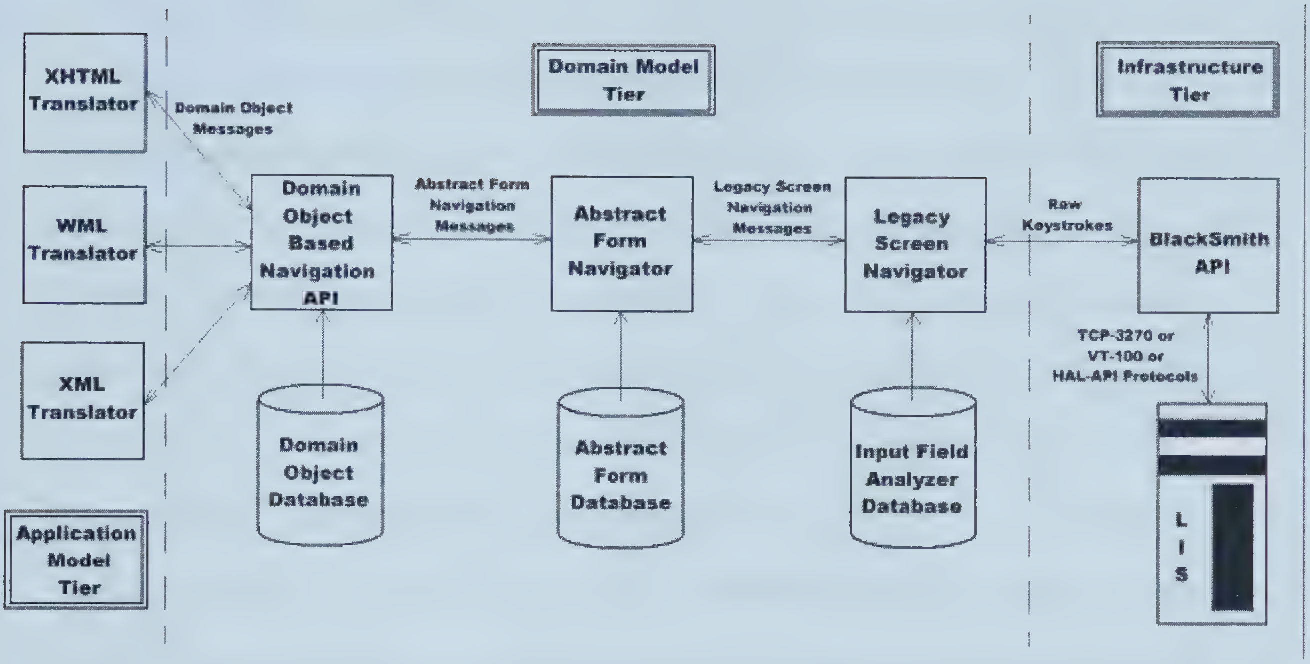


Figure 4.1: Mathaino Runtime Components

The base runtime layer is completely platform independent and consists of four main components. These are:

1. The domain model navigation subsystem.
2. The abstract form navigation subsystem.
3. The legacy screen navigation subsystem.
4. The BlackSmith terminal emulation API.

Figure 4.1 illustrates the design of the Mathaino runtime including its various constituent components.

As shown in Figure 4.1, each base runtime component forms a higher layer of abstraction over the previous component and, thus, offers a higher level of navigation API over the previous component. Once the user has input data using the platform specific GUI, the runtime translators export this data into the appropriate domain model semantics understood by the Domain Model Navigator. This component in turn utilizes the data produced during the domain-modeling phase of the Mathaino developer to translate various domain object fields into their corresponding abstract form fields. The Abstract Form Navigator again utilizes the data produced during the abstract GUI generation phase to translate these abstract form fields into appropriate screen navigation commands for the Screen Navigator component. Finally, the Screen Navigator component utilizes the datastore produced by the Input Field Analyzer component (see section 3.3.2) to translate each screen navigation command into its corresponding set of raw keystrokes for the BlackSmith terminal emulator, which then drives the legacy system.

Besides executing the task trace, these runtime components are also responsible for translating the output produced by the legacy system to an appropriate layer of abstraction. Thus, a similar reverse translation scheme is applied to each output field as it is passed along to the next higher level navigation component. For example, the Domain Navigator component annotates the output produced by the Form Navigator Component with the appropriate domain object field information before passing it along to the platform specific translators.

As explained in the previous chapter, Mathaino executes a given task trace on the legacy information system using pre-determined non-deterministic navigation plans (see section 3.3.4). The unit of navigation for this plan is a single abstract form produced by the Abstract GUI Generator plugin of the Mathaino developer subsystem. Since each abstract form itself comprises of various domain object fields, if the corresponding domain object fields for an abstract form are passed in then also navigation can commence. Thus, the highest navigation API possible in the base runtime is at the level of domain objects. The purpose of the domain navigator component is to provide a navigation API at this highest possible level of abstraction. This component is also used extensively by the XML translator component since that component performs navigation tasks for external applications using domain objects encoded in the XML notation. Since domain objects are platform independent in nature this API is platform independent too. It is the responsibility of the platform dependent translators present in the application model tier to translate the platform specific messages into the platform independent form required by the domain model navigator component.

After a domain object field has been resolved into an appropriate abstract form field, and the entire set of input fields for a given abstract form are available, the task execution on the backend legacy information system can commence. The purpose of the Form Navigator component is to provide a navigation API at the level of abstract GUI forms. Again, since abstract forms are designed to be platform independent this form navigation API is platform independent too. Thus, any component that needs to utilize the Form Navigator component is required to communicate the plan data to and from it in a

platform independent manner. Since the Domain Model Navigator component already encapsulates domain objects in a platform independent form, translating the domain object fields into appropriate abstract form fields is not a very convoluted task.

The most basic navigation API present in the entire Mathaino runtime is the BlackSmith terminal emulation API provided by Celcorp [BLACK98]. This is a very low-level navigation API that works at the raw keystrokes level. The primary advantage of constructing the Mathaino runtime on top of this API is BlackSmith's ability to offer a unified navigation API for a host of legacy information system protocols. BlackSmith currently supports TCP-3270, VT-100 and HAL-API communication protocols. However, since the BlackSmith works at the raw keystrokes level, a more abstract, higher level navigation API is required whose unit of navigation is an entire legacy system screen. This is needed because while the form navigator component can translate a given form representation into an equivalent set of screen navigation commands fairly easily, translating an abstract form database directly into a sequence of raw keystrokes required by the BlackSmith API would be quite cumbersome. The screen navigator component of the Mathaino runtime has been built on top of the BlackSmith API to provide this higher level navigation capability. Further, building a separate individual screen level navigation API is also useful for another component known as the Plan Navigator component which is used by the Mathaino developer at the development time to verify navigation plans. Since this component too requires a screen level navigation API, it too benefits from the Screen Navigator component of Mathaino runtime.

As pointed out by [MOORE96] abstraction is the key for successful legacy information system reengineering. As discussed before, the entire set of components in the base navigation runtime are platform independent and the primary unit of navigation is the set of domain objects for the given LIS. It is the job of the platform specific GUI translators present in the Application Tier to perform dynamic forward engineering at the runtime so that the platform specific GUI messages can be translated to a platform independent form understood by the backend runtime components.

Till now the only suggested technique for construction of multi-platform user interfaces has been via the use of abstract multi-platform GUI libraries [ENG96, NICHOLS91]. The attempt in this approach is to achieve portability by using a portable GUI toolkit. Also Melody Moore has suggested a knowledge-based approach for widget translation [MOORE94]. Again this approach focuses on code migration. For Mathaino we have experimented with a new approach where the GUI itself is designed to run on an abstract window toolkit. Hence, at runtime it can be interpreted on any platform that supports a translator for this abstract GUI specification. Essentially, this is the same idea as the Java virtual machine, extended to the domain of graphical user interfaces. Similar to Java bytecode, we compile our abstract GUI into a set of common abstract widgets specified in the XML notation. At runtime it is the responsibility of each platform specific translator to select a native widget that closely resembles the abstract widget specified by the abstract GUI. In this way we achieve multi-platform operation without introducing the complexities associated with code migration. Also we manage to provide a native look-and-feel as each runtime translator is free to interpret the abstract GUI in accordance

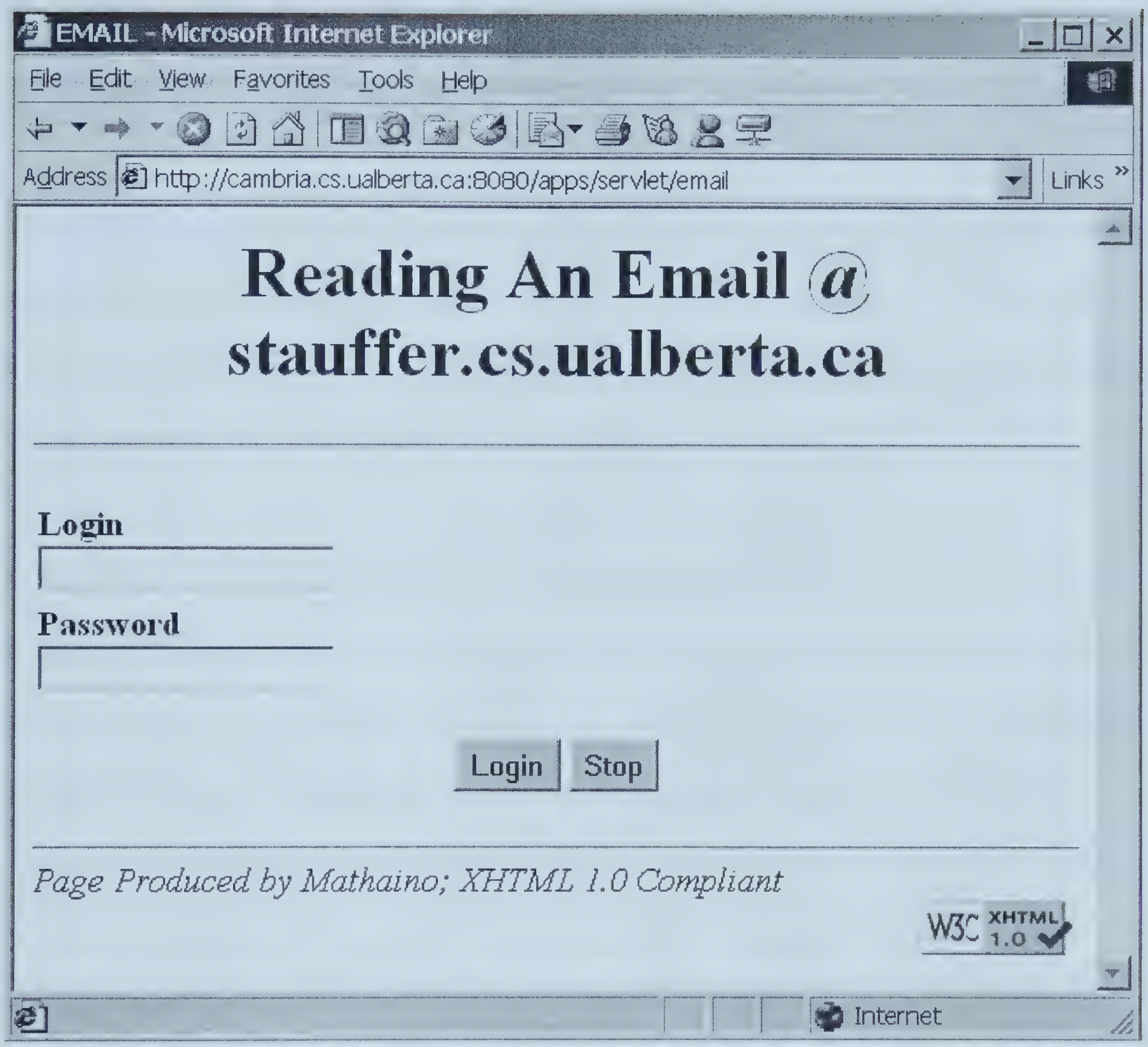
with the GUI standards for its specific platform. We have developed a set of three platform specific translators for Mathaino. These are explained further in the following sections.

4.1 The XHTML Translator

The XHTML translator is used for migrating a given legacy system to web-based platforms. The following sections explain the XHTML language and the XHTML translator in more details.

4.1.1 What is XHTML?

XHTML is the next generation of HTML. Currently XHTML 1.0 has been approved as a W3C recommendation [XHTML00]. Essentially XHTML is a reformulation of HTML 4.01 in XML. Thus, any XHTML document is also a valid XML document. As most Mathaino components, including the Abstract GUI Generator, produce content in XML, XHTML was a natural choice for implementing the abstract GUI on web browsers. Currently, both Microsoft's Internet Explorer (versions > 5.0) and Netscape's Navigator (Mozilla MI18 or Navigator 6 PR 3) can parse XHTML web pages. Thus, the XHTML GUI produced by this translator can be executed on any computer that is capable of running any one of these web browsers.



EMAIL - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address
Links »
Login
Stop
W3C XHTML 1.0
Internet

http://cambria.cs.ualberta.ca:8080/apps/servlet/email

Reading An Email @ stauffer.cs.ualberta.ca

Login

Password

Page Produced by Mathaino; XHTML 1.0 Compliant

Figure 4.2: An XHTML Interface for Logging into Pine

4.1.2 The XHTML Translator Explained

Mathaino's XHTML translator maps the abstract GUI forms to appropriate XHTML CGI forms. The translator has been built using the Java Servlet API [JAVAS00] and it executes as a servlet on the web server. It dynamically parses the abstract forms and translates them into XHTML at runtime. It also parses the CGI response produced by the

client web browser into the platform independent domain object field notation required by the Domain Model Navigator component.

The XHTML translator maps the abstract GUI widgets into appropriate CGI widgets. For example, the range fields are mapped to either a combo box widget or a set of radio buttons. The translator also uses XHTML tables to layout the web page in a format that closely resembles the format chosen by the user for the abstract GUI.

In this way, the XHTML translator can execute the abstract GUI on web browsers in a manner transparent to the end user. Web pages produced by Mathaino's XHTML translator have been validated by W3C as being 100% XHTML 1.0 compliant [VALID00]. Figure 4.2 illustrates an XHTML web page for logging into the Pine email reader.

4.2 The WML Translator

The WML translator is used for migrating a given legacy system to mobile web-enabled devices. The following sections explain the WML language specification and the WML translator in more details.

4.2.1 What is WML?

WML stands for Wireless Markup Language. As is obvious from the name, WML has been developed for rendering web pages on WAP enabled mobile Internet devices. The

WAP forum [WAPFOUR], which is a consortium of various wireless device manufacturers, has been responsible for standardizing this new markup language. WML is based on XML (eXtensible Markup Language). The DTD for WML 1.1 can be found at [WMLDTD00].

Both XML (which is the parent language for WML) and HTML are markup languages whose basic design is based on another markup language known as SGML. This leads to certain similarities between HTML and WML. For example, both these languages use simple tags for encoding the data. Also both these languages are encoded using plain text (i.e. characters whose value is less than or equal to 128 in the ASCII character set). However, the similarities between the two end here. For example, unlike HTML, a WML document must be well formed. This is necessary, as XML doesn't allow malformed documents. Further, as WML is derived from XML all WML documents must conform to a well-specified DTD. WML doesn't allow tags to be nested in an arbitrary manner as this may violate the DTD. This is a big difference from HTML where the order of tags within the document usually carries no importance. Also, unlike HTML, WML is case sensitive. Thus, the tags <b> and <B> are not the same in WML. Besides these, there are more fundamental differences between WML and HTML. These and other such restrictions are covered in the next section.

4.2.2 WML: Differences and Constraints

As discussed previously, WML is based on XML. This leads to some fundamental differences between WML and HTML. Some of the most obvious ones have already been

discussed in the previous section. In this section we highlight the other subtle constraints that WML imposes on mobile applications.

The main motivations for WAP forum for adopting a new markup language were the unique requirements of mobile devices. Unlike personal computers, mobile devices like HandHelds, Cellphones etc. have many hardware oriented limitations. Some of the main limitations of these devices are listed below:

1. Usually WML capable devices like HandHelds etc. have a very limited screen real estate. Thus, the user interfaces for these devices have to take into consideration the restrictions imposed by the limited size of the device's output screen.

2. Due to their miniature size these devices usually ship with either no or very limited secondary memory storage.

3. Further, the primary memory of these devices is quite restricted too. It is not uncommon for these devices to have less than 4 MB of RAM (This is more applicable for WAP devices like Cellphones than HandHelds).

4. Also the processing power of these devices is quite restricted.

5. And last but not the least, the power consumption for these devices has to be very low as they are expected to function for hours on their limited battery supply.

It is these hardware restrictions that drive the main design of the new markup language (WML) developed for them.

Unlike HTML documents that are organized in a flat tree manner, WML documents are organized in a two level hierarchy. The first level of a WML document is the WML deck. The second level consists of one or more WML cards that are stored inside this deck.

The main purpose of a WML deck is to act as a container for a set of WML cards. Also certain global attributes that should apply to each card in the deck are encoded in the parent deck. As each deck is a valid XML document, it is coded in accordance to XML rules. Thus, all the tags inside the deck can be organized in a single rooted tree. Each WML deck consists of one or more WML cards. It is these cards that are actually displayed to the user. Only a single card can be displayed to the user at a time. Thus, even though WML cards are the units of display on the WML device, the unit of communication between a WML device and a WML capable server is a single WML deck. This scheme is adopted as transmission of multiple cards in a single request is more efficient and has implications on the battery consumption of the WAP device. This arrangement of decks and cards is something that is unique to WAP devices and has no corresponding counterpart in HTML.

WML applications can communicate data back to the WML server, providing a facility almost similar to CGI for WML devices. However, WML has no concept of forms like

HTML does. All WML CGI data must be sent back using the `<postfield>` WML tag, which sends back each WML field individually to the backend server. This is quite different from the scheme adopted in HTML where the HTML form embodies the entire CGI transaction. Also, as the concept of a form does not exist in WML, the data input section of a WML card does not look anything like the input section of an HTML form. The data input sections in WML are completely unrelated to the `<postfield>` sections where the data is actually sent back to the backend server. This design difference makes the process of coding a WML CGI application quite different from that of an HTML CGI application.

Unlike HTML, which has no concept of variables, WML supports inline variables. In fact, any data input to a WML CGI application is stored in a WML variable. The `<postfield>` expression can be used to post any arbitrary value back to the server, including variables. Thus, the concept of variables combined with the facility provided by the postfield expression allows us to simulate HTML CGI forms in WML, even though no direct translation for HTML forms exists in WML.

Also, WML does not allow any JavaScript code. A separate scripting language called WMLScript exists for WML applications. WMLScript is not compatible with JavaScript.

The most significant constraint on WML applications is imposed by the WAP protocol. The WAP protocol does not allow transmission of more than 1200 bytes of data in a single request. Since a WML deck is the unit of communication between a WML device

and the backend WML server, this effectively restricts the maximum size of any WML deck to 1200 bytes. This has significant implications for any WML application designer as it means that he can not transmit any web page that contains more than 1200 bytes of data in one go. As legacy applications recognize no such restriction, Mathaino has to somehow honor this size restriction, while migrating them to the WML platform.

To summarize, the various differences restrictions between HTML and WML platforms are highlighted below:

1. The WML document model is radically different from the HTML document model.

2. WML has no direct support for CGI forms, although, it does provide features that can be utilized to simulate this facility in WML.

3. Unlike HTML, WML has built in support for inline variables, and expects any WML CGI application to utilize it.

4. WMLScript and JavaScript are not compatible.

5. The maximum size of any WML deck (or transmission) can be no more than 1200 bytes.

4.2.3 The WML Translator Explained

In this section we describe the WML translator subsystem of Mathaino. As discussed previously, Mathaino's base runtime subsystem is platform independent. The two most important subsystems in this scheme are the Abstract GUI Generator and the Domain Model Navigation API. The Abstract GUI Generator of Mathaino produces a set of abstract forms that represent the new user interface for the legacy system. The platform specific translators are responsible for rendering this GUI on the target platform and translating back the platform specific GUI messages into the form understood by the Domain Model Navigation subsystem, which is again platform independent. In order to connect to a web server on the Internet the WML devices use HTTP tunneling to transfer WML data from WML capable web servers to the WAP service provider. This enables any web server that can upload WML content to talk to WAP enabled devices.

An abstract form produced by the Abstract GUI Generator primarily consists of two portions. The first part of the form consists of the output area and the following portion consists of the input area.

The output area of an abstract form contains various output fields that encapsulate the output produced by the legacy system. Similarly, the input area consists of input fields that represent the data to be input to the legacy system. It is necessary to preserve this order while translating an abstract form to the WML platform as the user needs to see output from the legacy system to deduce the values for the following input fields. Since this order has to be preserved, the WML translator of Mathaino translates each abstract

form into two corresponding WML decks, the output WML deck and the input WML deck. Further, as we have no control over the length of the output data produced by the legacy system, and the maximum permitted size of any WML deck is 1200 characters, the WML translator generates multiple output decks in case the amount of output produced by the legacy system exceeds this limit. Each one of these output decks is then sent to the WAP device in sequential order. In this manner, we spread the entire output over multiple decks in order to transmit it to the user. The input decks for the abstract form follow the output decks. Again, depending on the number of input fields, multiple input decks might be generated for a single abstract form. Pictorially this scheme is illustrated in Figure 4.3.

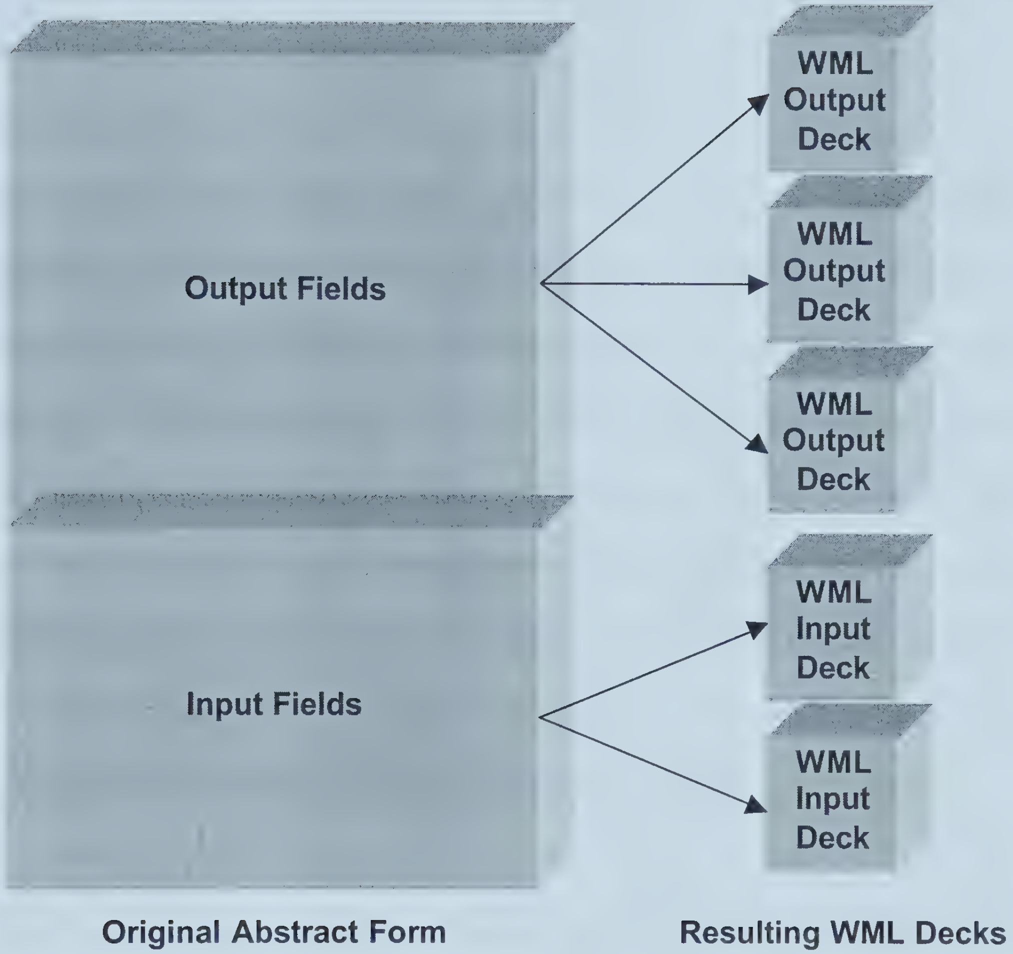


Figure 4.3: Dividing an Abstract Form into multiple WML Decks

The WML translator is also responsible for mapping the set of abstract input widgets into their platform specific counterparts. Currently, the abstract window toolkit of Mathaino provides the following set of input widgets:

1. Edit Box
2. Password Box
3. Combo Box

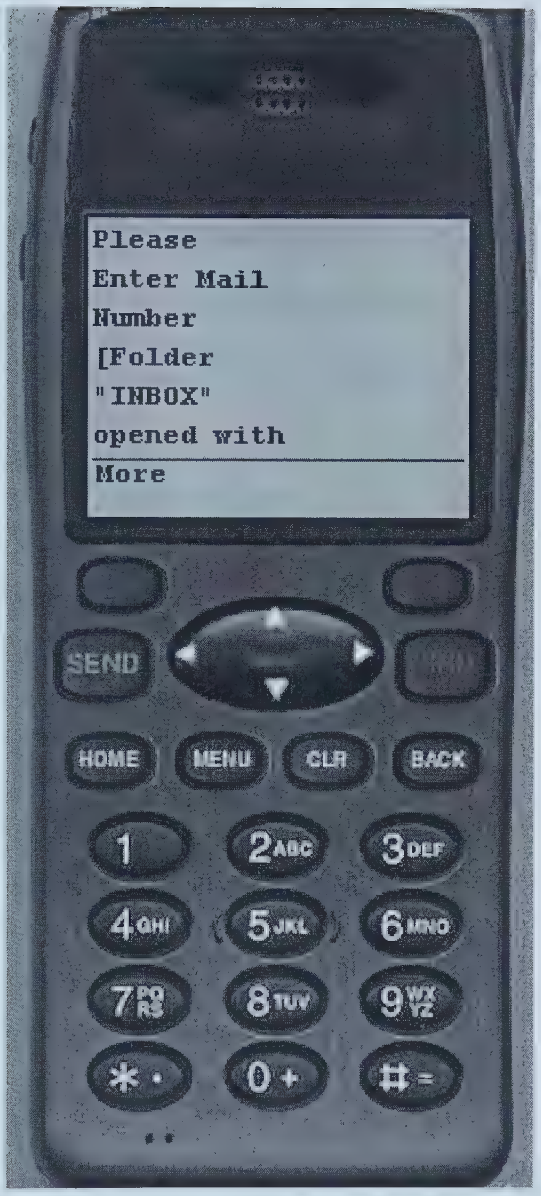
4. Radio Button

The WML platform does not provide either the combo box or the radio button widgets. Further, in order to group multiple input widgets in a single card it is necessary to group them within the `<select>` WML tag. This organizes the set of input fields in a simple numerical menu. Such an arrangement is necessary as the WML translator tries to render the abstract form on the WML device as accurately as possible. The WML translator also uses the `<select>` WML tag for displaying both the combo box and radio button widgets. To achieve this, it lists all the choices encapsulated by these widgets in a simple numerical menu, prompting the user to select any one of these. In this way, we can effectively simulate the original set of widgets on the WML device.

The WML translator maps each input widget into a separate WML variable. It also generates the appropriate WML code necessary to post the values of these variables back to the web server.

While this scheme solves the problem of rendering the abstract form on the WML device, it creates a new problem for the backend navigation subsystem. This is because the backend navigation subsystem of Mathaino relies on the unified notion of an abstract form. It expects the entire form to be rendered in one request and the entire input data to be available as a single response. As the input portion of a single abstract form may be divided into multiple WML decks, the entire input data for that form may not be available in a single response. Clearly re-implementing the backend navigation subsystem is not a

solution to this problem as that invalidates the notion of platform independence of the basic Mathaino runtime subsystems. To overcome this problem the WML translator caches the user input from the multiple decks locally before sending it to the backend navigation subsystem. This makes the process of form division transparent to the backend subsystem.



Please
Enter Mail
Number
[Folder
"INBOX"
opened with

More

Figure 4.4: A WML input card for the Pine Email Reader

In this way, the WML translator allows the user to use the legacy system from this WAP enabled Internet device. Figure 4.4 illustrates one of the input decks for an abstract form for the Pine email reader being rendered on a WML enabled Cellphone.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE task SYSTEM "task.dtd">
<task tasktype="stauffer.cs.ualberta.caRead_An_Email">
  <inputfieldalias>
    <info type="User">
      <name>Login</name>
      <xpath>User.Login</xpath>
      <value>kapoor</value>
    </info>
    <info type="User">
      <name>Password</name>
      <xpath>User.Password</xpath>
      <value>abcxyz</value>
    </info>
    <info type="Email">
      <name>ReadCommand</name>
      <xpath>Email.ReadCommand</xpath>
      <value>ij1</value>
    </info>
  </inputfieldalias>
  <outputfieldalias>
    <info type="Email">
     <name>Date</name>
     <xpath>Email.Date</xpath>
     <value>28 Apr 2001 16:08:23-0600(MDT)</value>
    </info>
    <info type="Email">
      <name>From</name>
      <xpath>Email.Sender</xpath>
      <value>R. Kapoor <kapoor@cs.ualberta.ca></value>
    </info>
    <info type="Email">
      <name>Subject</name>
      <xpath>Email.Subject</xpath>
      <value>Subject: Test</value>
    </info>
      . . .
  </outputfieldalias>
</task>
```

Figure 4.5: A Mathaino Task Model

4.3 The XML Translator

Mathaino runtime also includes an XML translator. The main motivation for creating this translator was to enable external applications to integrate and interoperate with LIS

wrapper applications generated by Mathaino. The XML translator utilizes domain objects encoded in XML notation for its operation.

An XML task specification for Mathaino is encoded using a task model notation. A Mathaino task model consists of a set of domain specific objects that participate in the execution of that particular task. As can be seen from the fragment task model of Figure 4.5, the elementary interactions at the infrastructure tier (please refer to Figure 4.1) have been transformed to input/output information in the application domain. As discussed previously, the application domain for Pine is very simple: a user class, containing two attributes, login and password, and an Email class containing four attributes, a date, a sender, a subject, a message, and a ReadCommand method. At the infrastructure tier, the data manipulated through the elementary interactions is not typed; their only attribute described in the dialogue model is the location of the activity where the input/output interaction takes place. After having provided a domain model to Mathaino, these data elements become instances of a class model with rich relations between them. For example, from Figure 4.5, it can be seen that the values "Kapoor" and "abcxyz" are the login and password of a single user.

Once a complete task specification has been forwarded to Mathaino using this XML task model, it can independently carry out that task and also export the results produced in a similar XML notation, utilizing the existing domain model for that legacy information system.

Within the Cellest project, we have also developed a mediator application known as Babel [ZHANGSTR01] that can communicate with Mathaino using Mathaino's task models encoded in the XML notation. Thus, the XML translator not only allows external applications to drive Mathaino generated LIS wrapper applications but also allows for a standardized means for data import and export from them. Hence, the XML translator leads to an increase in the amount of code reuse and interoperability.

Another set of components that constitute the Mathaino runtime are the output field parsers. These are being discussed here for the sake of completion. As discussed in Chapter 3, Mathaino has two sets of output field parser families: the coordinate invariant parser family and the coordinate variant parser family. The runtime contains parser plugin components for these two output field parser families that are capable of parsing various output fields form incoming legacy screens using the templates that have previously been generated by their developer counterparts.

Sequence Number	Screen Description	Input Action Item	Output Produced
1	Login Screen, prompts for Login Name	Login name entered	
2	Login Screen, prompts for Login Password	Login password entered	
3	Solaris console screen, contains UNIX prompt	`set term=vt100`	
4	Solaris console screen, contains UNIX prompt	`pine`	
5	Pine Main Screen	`i`	Total Number of emails in the Inbox
6	Pine Inbox screen	`j + <Message To Read>`	List of Emails in the Inbox
7	Pine Message Screen	`qy`	Email Date, Subject, From and Message
8	Solaris console screen, contains UNIX prompt	`exit`	

Table 4.1: Legacy Pine Screens Encountered while Reading an Email

4.4 An Example Task Trace: Reading An Email Using Pine

In this section we will demonstrate the functionality and the roles of each runtime component by running through a task trace for reading an email using the Pine email reader. We have been using this task trace as an example since Chapter 3, in this final section we will demonstrate how the various runtime components cooperate in order to execute this task on the migrated platforms.

As pointed out previously, this task trace was recorded using a Pine email reader program hosted on a Sun Solaris server at the Department of Computing Science, University of Alberta. For the purpose of this example, we assume that the appropriate developer data structures have already been generated and verified for the given task trace and an acceptable abstract GUI has been generated for it using the Abstract GUI generator developer component.

This task consists of a sequence of eight legacy system screens. The role of each legacy system screen along with the entire set of input information supplied to it by the expert user is listed in Table 4.1.

As pointed out in Chapter 3, for this task, at the termination of the analysis phase the total set of input information required from the actual user had been reduced to just three input fields i.e. the user login, password and the serial number of the email to read. The complete set of domain objects constructed for this task are explained below:

Domain Object Field	Legacy System Field
User::Login	Screen 1, Login
User::Password	Screen 2, Password
Email::ReadCommand	Screen 6, Email Number
Email::From	Screen 7, Email From
Email::Date	Screen 7, Email Date
Email::Subject	Screen 7, Email Subject
Email::Text	Screen 7, Email text

Table 4.2: Domain Object Field to Legacy System Field Mappings

1. **Domain Object "User"**

   This object represents the user whose email needs to be read. It contains the following two fields:

   a. User::Login: This contains the user's login name.

   b. User::Password: This contains the user's login password.

2. **Domain Object "Email"**

   This domain object represents a single email. Its fields and methods are discussed below:

a. Email::ReadCommand: This methods takes the serial number of the email to be read as its parameter.

b. Email::Subject: The subject of the email.

c. Email::Date: The date on which this email was sent.

d. Email::From: The sender's name and email address (if available).

e. Email::Text: The message content of the email.

Each one of these object fields maps to a unique input or output field for this task on the Pine legacy system. Table 4.2 lists the mappings between these domain object fields and the legacy system fields.

Further each abstract form groups a sequence of these legacy system screens and their respective navigation plans. The set of abstract forms generated for this particular task are explained below.

1. **The Login Form:** This form encapsulates the entire user object. It is used to log into the Pine email reader program. This form is the first form encountered during the enactment of this task trace.

2. **The Read Command Form:** This form contains a single input field i.e. the number of the email to be read from the list of emails currently available in the Inbox. Since the user needs to know the number of unread emails currently waiting to be read in

Abstract Form	Purpose	Screens Grouped	Domain Object Fields
Login Form	Login to Pine	Screens 1 to 5	User::Login, User::Password
Read Command Form	Select Email to be Read	Screens 5 to 7	Email::ReadCommand
Message Form	Show Selected Email	Screens 7 to 8	Email::Date, Email::From, Email::Subject, Email::Text

Table 4.3: Grouping Legacy Screens into Abstract Forms

his Inbox before he can make this decision, this form also incorporates an output fields that lists the total number of unread emails in his Inbox.

3. **The Message Form:** This form displays the contents of the email selected by the user in the previous form.

Therefore, the sequence of eight LIS screens required for the enactment of this trace is reduced to a set of just three abstract GUI forms. Table 4.3 illustrates the grouping of the legacy system screens for each one of these abstract forms.

As discussed previously, at runtime the GUI translators are responsible for rendering these abstract forms and also translating platform specific messages into the abstract notation understood by the base Mathaino runtime subsystem. The sequence diagram

shown in Figure 4.6 illustrates the entire enactment process of this trace, along with the appropriate input and output messages generated at each component layer.

As shown in the Figure, the messages from the GUI Forms are translated by the platform specific translators into their appropriate Domain Model fields before sending them to the Domain Model navigator which then re-translates them into the appropriate form for the Abstract GUI Navigator. The LIS Screen Navigation component has access to the Input Field Analyzer database, thus, each abstract form can cause multiple legacy screen transitions as many screen navigation commands can be automatically filled in by the LIS Screen Navigator using the Input Filed Analyzer database generated during the analysis phase of the Mathaino Developer. Thus, the set of three abstract forms, and their associated set of three input, and five output fields are sufficient to navigate the entire PINE trace.

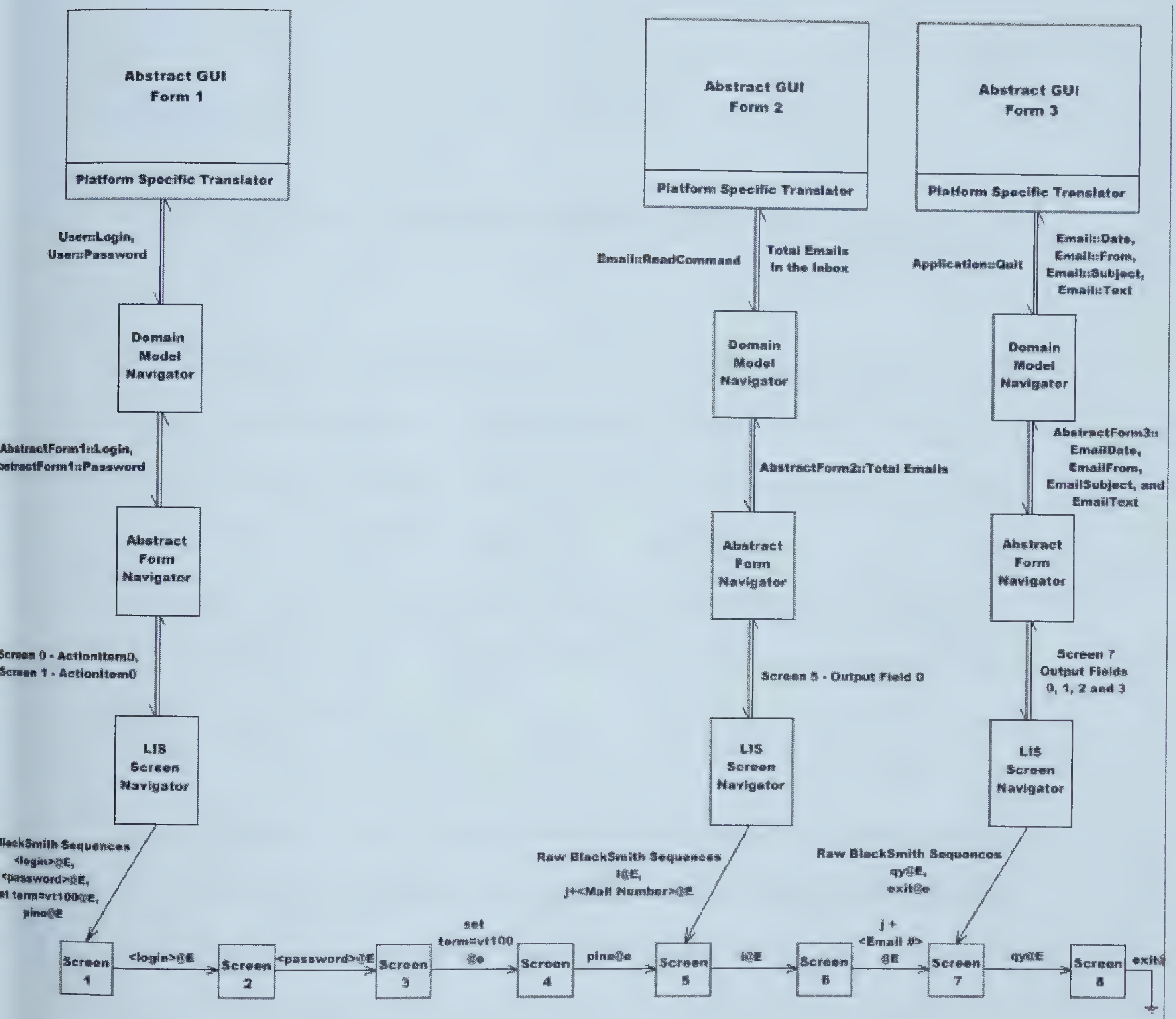


Figure 4.6: Executing the Pine Task Trace

Chapter 5

Mathaino Development Methodology

The CelLEST group, of which Mathaino is a constituent, functions under the aegis of the Software Engineering Research Laboratory (SERL) at the University of Alberta. Being software engineers ourselves, research into core software engineering technologies, especially the hot topic of software process models is quite close to our hearts. Consequently, one of the minor research goals of the Mathaino project was to act as a test bed for research into the emerging field of lightweight, goal oriented and experimental software process models.

Back in April 2000 when we initiated the development process for Mathaino, one of the highly controversial (see [SIDDIQI00]) but quite promising software process technique was Kent Beck's eXtreme Programming (XP) [BECK99]. As mentioned before in this thesis, the pre-runner for Mathaino was the Urgent Project [KONGSTR99, STROULIA99]. While Urgent had met its research aims, our goal for Mathaino was to significantly improve upon Urgent's capabilities and design Mathaino to take advantage of what we had previously learnt about this domain from the Urgent project.

Since both Mathaino and Urgent shared their application domain, initially our aim was to reuse significant portions of Urgent's code base so that from the very start we could concentrate on extending and evolving Urgent instead of recreating its existing abilities from scratch. However, after a detailed inspection of the Urgent's code base it became quite clear to us that due to typical problems associated with traditional up front design paradigms, the project specifications and documentation for Urgent was out of sync with its source code. Further, as traditional software process models lay more emphasis on "*paper rather than people*" [MARTIN00] and fail to lay adequate emphasis on simple yet highly beneficial coding techniques like code refactoring, the condition of Urgent's code base was not in a position where it could be readily extended or even comprehended. We had a clear choice, we could either spend more time and resources, both of which are always short for research projects, on understanding and cleaning Urgent's existing code base or completely scrap it and start from scratch.

The main disadvantages of starting the development of Mathaino from scratch were:

1. As pointed out previously, we would have had to redevelop even existing Urgent functionality.

2. After that we would have had to start extending it. This is when the actual productive phase (from the point of view of research) for Mathaino would start.

3. Finally, Mathaino's new code base could itself end up being as bad as Urgent's probably requiring the next developer to start from scratch again.

Even though it seemed like taking a step backwards, we decided to redevelop Mathaino from scratch, while borrowing the successful algorithms from Urgent for the pieces of code that we would have to redevelop. Existing research into software development has shown that redeveloping from a cleaner code base actually leads to overall faster development times than developing on top of a convoluted code base [FOWLER99]. This was our prime motivation for deciding in favor of redevelopment. Having decided upon this choice, it was now obvious that:

1. We needed Mathaino's development to proceed at an express rate if we were to actually redevelop and extend Urgent and still finish within the allotted time (approximately one year).

2. We needed to make sure that Mathaino's resulting code base would be of significantly improved quality over Urgent's so that the next Cellest developer for this domain wouldn't have to restart from scratch again. Thus, we needed to significantly improve code quality while also significantly improving the development speed! Clearly these two requirements were orthogonal to each other as far as tradition software process models were concerned.

It were these requirements, coupled with our previous unsatisfactory experience with traditional software process techniques that first ignited our interest in the rapidly emerging field of eXtreme Programming (XP).

As we conducted more research into XP, we were able to further substantiate our initial suspicions that it might prove to be a very successful set development techniques, specially for the domain of academic research projects. This is because risk is a significant factor during the development of research projects and the highly iterative development process of XP is especially efficient at mitigating risk.

In the next section we highlight the issues in research projects that specifically make XP quite an obvious choice for them as far as software process models are concerned.

5.1 Software Engineering Issues In Research Projects

Research projects usually operate in a significantly different environment than traditional software development projects. First, by their very definition, research projects are experimental in nature and are aimed at previously untested technologies. This makes them a highly risk intensive activity. It is obvious that because of their unique environment, traditional software development models are even less suited to the domain of research projects than they are to more traditional software projects, like commercial software development. Some of the main software engineering issues for research projects are discussed in the following sub-sections.

5.1.1 Clean, Understandable Code

It is almost guaranteed that the original set of developers for a research project (who, of course, happen to be graduate students) will not be available later on (i.e. after they have graduated). Also usually a research project will need to be extended even after it original

developer has left (i.e. as and when its principal researcher i.e. the supervisor smells another set of interesting problems that need to be resolved within the same domain). Thus, it is very important that, at the very least, the source code of a research project be easily understandable to the new developers (i.e. the next set of graduate students who will work on it).

5.1.2 Small Team of Developers

It is well known that research projects have a very small team of developers (usually it is restricted just to the supervisor, the research student and probably a couple of research associates).

5.1.3 Restricted Set of Resources

Research projects are also carried out under very tight resource and time constraints. Usually the principal researcher, who is the project supervisor, has a fixed grant for a given research project. Its development cost must not exceed this grant. On the other hand the principal developer, who usually is the graduate student, sets a certain fixed time constraint for themselves and would like to stick to it as far as possible. Thus, scheduling and measuring project velocity which are two of the biggest weaknesses of traditional software control processes are of prime concern in research projects.

5.1.4 Constant Evaluation

Also, at least theoretically, the supervisor would like to would like to be in a position where they can observe concrete work (not just theoretical design) done at any particular stage of development. This is quite important as in more cases than not, the supervisor

has to submit regular progress reports to the sponsoring agency for that research project so that they can justify the work done for the resources consumed so far by the project. Since usually the sponsors of a research project like to be able to see actual progress coupled with concrete results (for example, we have implemented functionality X, Y and Z with the expected results A, B and C being verified or not verified) instead of just proposed hypothesis (we have completed analysis for stage X, Y and Z and will proceed to the design phase next and will report on the results or failures at the end of the research grant's term), constant demonstrable evaluation of progress becomes quite an important issue.

5.1.5 Fluid Requirements

Research projects have a very fluid set of starting requirements, even more so than traditional software development projects. This is because the problem that they are targeting is, by definition, an unclear, research problem whose solution might change dynamically with the progress of the project as newer issues and insights are discovered and resolved.

5.1.6 Easy Extension

Since previously unknown factors will almost definitely emerge and affect the progress of the project with time, the current direction of a research project might have to be significantly altered requiring a non-trivial amount of design and code changes. In addition to this, previously unplanned set of features might need to be implemented in a cost and time efficient manner.

5.1.7 Reliable Code

Correctness of code throughout development is a prime requirement for research projects. If the source code of a research software solution is suspect then none of its current set of conclusions can be treated as reliable, and thus, the whole project is deemed to be a failure. Thus, research projects are even more sensitive to code bugs than traditional software development projects.

5.2 eXtreme Programming to the Rescue

The very issues that make traditional software process models inappropriate for the domain of research projects make extreme programming an almost perfect fit for them. We list the specific strengths of XP that relates to each one of the issues affecting research projects that we have highlighted in the previous section.

5.2.1 Clean, Understandable Code

According to the XP philosophy, code and coders are the most important aspects of any software development project [MARTIN00]. XP emphasizes on collective code ownership. Thus, even if the original set of developers leave, the code base of an XP project would still be completely understandable to the remaining personnel. Further, since XP has a highly disciplined philosophy as far as code development is concerned and lays an amazing amount of stress on coding in accordance with strict project standards, the resultant code base of an XP project is simple, clean, highly efficient and easily comprehensible [DYNABOOK].

Within the Mathaino project we have practiced collective code ownership by making sure that at least one research associate, who we are convinced will be around, at least, till the next set of developers commence work, not only understands but also actively participates in the development of the project. Also even before commencing the development for Mathaino, the entire Cellest group agreed upon a fixed set of coding conventions (see Appendix B) to ensure that the entire code base of Mathaino would exist in a standardized form that could be comprehended by anyone who was cognizant of these coding standards.

Iterations		
1	28, July 2000	Screen Parsers Ready
2	8, August 2000	Analyzer Ready
3	17, August 2000	Plan Navigator Ready
4	6, September 2000	GUI Generator Ready
5	27, September 2000	XHTML Translator Ready
6	27, October 2000	WML Translator Ready
7	22, January 2001	Non-deterministic Navigator Ready
8	9, February 2001	XML Navigator Ready

Figure 5.1: Major Mathaino Iterations

5.2.2 Small Team of Developers and Restricted Resources

The extreme programming methodology is specially suited for smaller teams of developers. Since the project developers are also the designers and architects in XP [NEWKIRK00, BECK00], this methodology is a good fit for small development teams that are commonly found in the domain of research projects. Also large development teams are actually unsuitable for experimenting with XP [SPINELLIS00].

5.2.3 Constant Evaluation

Since research projects have limited teams and resources and still require a functioning prototype at a very early stage XP's highly iterative development process with extremely short iteration times (usually 2 weeks) is a nice fit for this problem [FRASER00]. For example, an XP project can be halted at any time while still providing a functioning

system that reflects the investment till date into the project [MARTIN00]. Figure 5.1 illustrates the major iterations for Mathaino.

5.2.4 Fluid Requirements

An XP project doesn't have to adhere to a pre-determined static set of global requirements [MARTIN00]. Instead, the requirements are decided on a per-iteration basis i.e. the current set of requirements for an XP project is only applicable for the current iteration. Further, according to the XP philosophy, the cost of changing the system requirements remains flat throughout the development process, unlike the traditional software process models where it is estimated that it rises exponentially [SIDDIQI00]. Thus, an XP project can dynamically adapt to fluid requirements of research projects with a minimum impact on the development cost.

5.2.5 Easy Extension

Since XP projects can easily adapt to a change in direction and requirements, they are fairly malleable and new set of features can be added to them with relative ease.

5.2.6 Reliable Code

Perhaps the single most appealing factor for adopting XP in research projects is the requirement for correctness of code. Studies have shown that pair programming, which is one of the prime tenants of XP, significantly reduces the amount of bugs in the code [WILLIAMS00]. Further, continuous integration coupled with frequent, comprehensive testing, both of which are included in the set of XP's commandments, also significantly

increase code quality [KARLSSON00]. This again makes the XP development methodology a good fit for research projects.

It was due to these observations that we decided to experiment with extreme programming as the development process for Mathaino. In the following sections we first briefly introduce the concepts of refactorings based development and then highlight the refactoring centric development process followed by Mathaino. In the penultimate section of this chapter we will present some interesting statistics we observed while experimenting with extreme programming.

5.3 Introduction to Refactorings Based Development

Traditional life-cycle models identify maintenance as a distinct activity that takes place after a software system has been delivered and deployed. During that phase, local changes can be made to the system in order to eliminate bugs that have not been discovered during testing. In addition, more extensive changes may be made to the system design to improve performance or readability or even further maintenance activities in the future. Finally, even drastic rewrites are possible at this phase when new requirements arise that cannot be fulfilled by local changes to the original system design.

Maintenance activities are, thus, classified as

- **Corrective**, when their objective is to eliminate a defect, or

- **Adaptive**, when their objective is to enable the system to run in an evolved environment, such as new operating system or hardware, or

- **Perfective**, when the modifications add new capabilities to the system in response to new evolved requirements by the user.

Experience has shown that perfective maintenance activities are the most time consuming. They account for up to 65% of all maintenance costs, thus, developers will often adapt the system in the absence of any request for enhancement, in order to improve the system design so that anticipated future enhancements will be easier. This activity is known as *preventive* maintenance.

Lehman's laws [LEHMAN00] capture some interesting properties of software systems as they evolve under active, long-term maintenance:

1. **The law of continuing change:** In order for a system to continue being useful (and used) in the real world it must change continuously.

2. **The law of increasing complexity:** Modifications usually increase the complexity of the system. Resources must be allocated to preventive maintenance activities, in order to manage this complexity increase.

3. **The law of large program evolution:** Program evolution is a self-regulating process. System attributes such as size, time between releases, and the number of reported errors are approximately invariant for each system release.

4. **The law of organizational stability:** Over a program's lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.

5. **The law of conservation of familiarity:** Changes between releases are incremental and do not fundamentally impact the familiarity of the maintainers with the system.

The rate of requirement evolution has been continuously increasing, as software developers compete to provide some additional distinct qualities in their systems to distinguish them from the systems of their competitors. Furthermore we have been recently witnessing a proliferation of new platforms, on which many existing systems must migrate. Finally the average "*time-to-deliver*" has become shorter, due to increased market competition.

All these phenomena, have caused the nature of software development to shift from traditional waterfall [ROYSE70] and spiral [BOEHM88] models to more incremental development processes, such as extreme programming [BECK99].

Refactorings are behavior preserving code transformations that bring about local design changes. As each such transformation is behavior preserving, it also has a corresponding

inverse transformation. At any point in time, many alternative and even inverse refactorings may be possible. Which one to apply is left to the system developer, who has to make this decision on the basis of the future extensions anticipated for the system: the refactorings applied should evolve the system design in a way that further development will increase the overall complexity as little as possible.

It would seem that Lehman's laws should still apply in the context of refactoring-based software development. Refactorings can be viewed as a form of preventive maintenance, in that they are meant to decrease the complexity of the system design in anticipation of increases of complexity that extensions will bring about (law 1). Refactorings are local transformations and as such they do not affect the developers' familiarity with the system (law 5).

As discussed previously, in our group we have been examining the problem of software architecture evolution and its impact on architecture quality [STROULIA01] and have recently started to follow refactorings-based development in some of our projects. At the same time, we have also started to examine how the various software metrics evolve across the different system versions produced by this development process. The following sections further discuss the refactorings based development of Mathaino in accordance with the concepts of extreme programming.

Build	Type	Date	New Features
3	First Iteration	28/7/2000	Developed the plugins for recognizing user activity at fixed or dynamic coordinates of the system screen
13	Second Iteration	8/8/2000	Input Field Analyzer Successfully Tested (acceptance test)
19	Second Iteration	12/8/2000	Trace analysis capabilities added
20	Refactoring	6/8/2000	Unified the plugin architecture, cleaner design Lines of Code Reduced: 2183
21	Third Iteration	17/8/2000	Added plan navigator component (unit test)
35	Fourth Iteration	6/9/2000	Added the abstract GUI generation component (acceptance test)
37	Acceptance Test Failed	8/9/2000	Plan navigator test failed (acceptance test), project halted
38	Acceptance Test Cleared	10/9/2000	Plan navigator test passed (acceptance test), project resumed, Proxy server for BlackSmith added
43	Fourth Iteration	18/9/ 2000	Plan navigator developed and tested
44	Refactoring	9/9/2000	XHTML form navigator split into a generic FormNavigator base component and the XHTML form navigator component built on top of it. Lines of Code Reduced: 490.

Build	Type	Date	New Features
45	Refactoring	21/9/2000	`XMLizable` Interface refactored into a sub-interface of the Java `Serializable` interface.
52	Fifth Iteration	27/9/2000	XHTML GUI Translator test passed (acceptance test)
65	Sixth Iteration	27/10/2000	WML GUI Translator test passed (acceptance test)
67	Maintenance	12/12/2000	Minor Maintenance fix after acceptance test on Celcorp's facilities was passed (application acceptance test)
83	Feature Addition	1/1/2001	Screen recognizer component added (acceptance test)
85	Refactoring and Seventh Iteration	22/1/2001	Mathaino base navigator and related navigation classes refactored to support branch splits that will be encountered during non-deterministic navigation plans.
87	Eighth Iteration	9/2/2001	XML Task Execution engine added

Table 5.1: The evolution of Mathaino

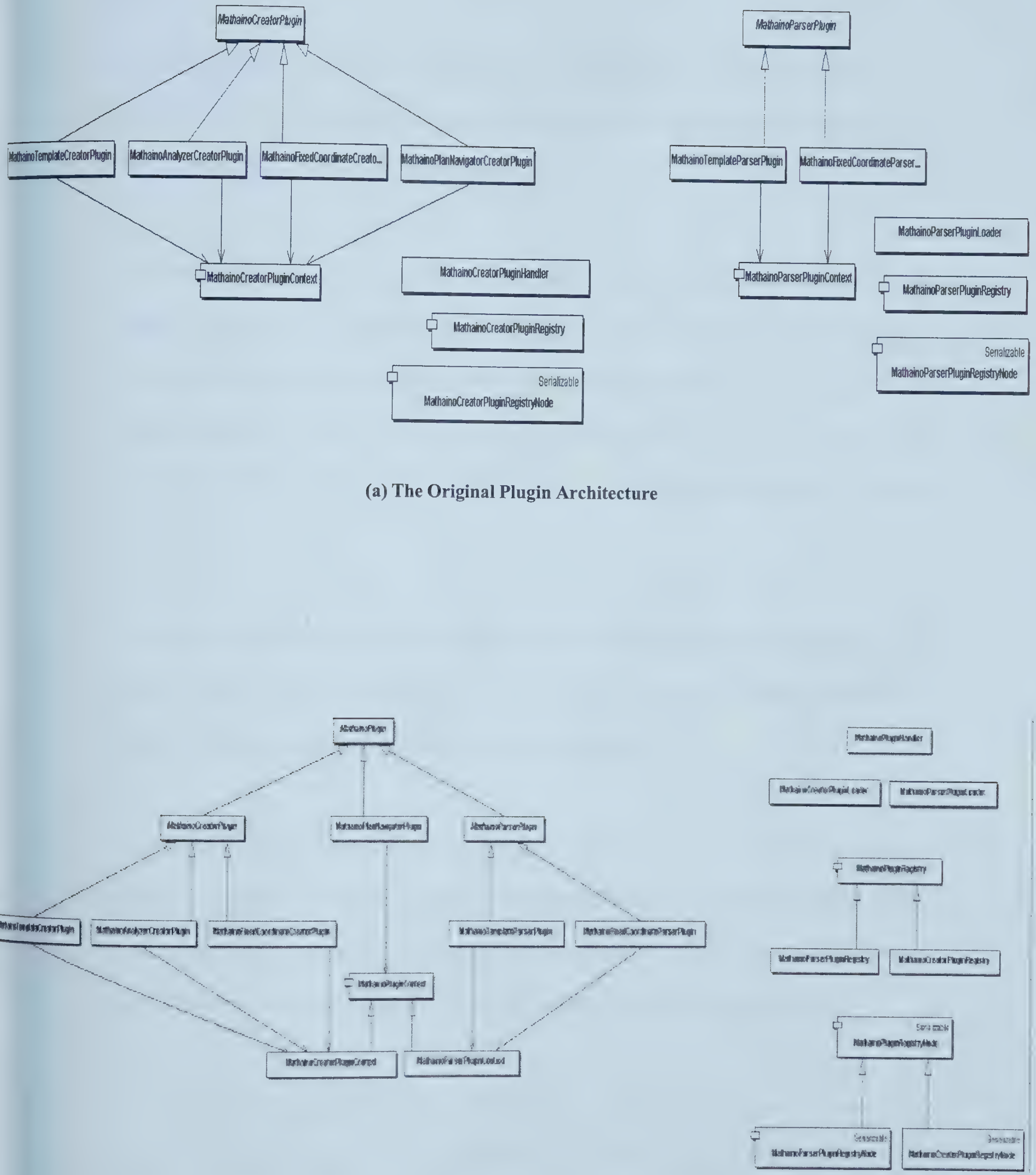
5.4 The Mathaino Case Study

Mathaino was developed by a single developer who has been following a strict refactorings-based development process, inspired by the Extreme Programming methodology. Active development for Mathaino started in July 2000 and until February 2001 the system had gone through eighty-seven builds. The activities between builds fall in one of three categories: development of new features, debugging after failures at acceptance testing, and refactoring.

Table 5.1 describes some major milestones in Mathaino's development. The first build mentioned is one of the earliest versions and is basically a re-implementation and, to a small degree, an extension of Urgent. Between this early build and the last reported one, we have selected four versions of the system which are the *before* and *after* versions of two interesting and quite consequential refactorings that we discuss in detail in the next two subsections.

5.4.1 Extracting a Common Superclass

As discussed in Chapters 3 and 4, Mathaino consists of a design-time and a runtime environment. The design-time Mathaino components analyze traces of the system-user interactions to construct model of the dialog and the locations of the interaction on the different system screens. The runtime Mathaino components use this model to parse the information provided by the original system interface and to forward it to the new front end. Clearly there is a direct correspondence between the components constructing the model and the components using it, however till its nineteenth version, Mathaino had two separate plugin hierarchies, one for the *developer* (i.e., the design-time components) and


MathainoCreatorPlugin
MathainoTemplateCreatorPlugin
MathainoAnalyzerCreatorPlugin
MathainoFixedCoordinateCreato...
MathainoPlanNavigatorCreatorPlugin
MathainoCreatorPluginContext
MathainoCreatorPluginHandler
MathainoCreatorPluginRegistry
Serializable
MathainoCreatorPluginRegistryNode
MathainoParserPlugin
MathainoTemplateParserPlugin
MathainoFixedCoordinateParser...
MathainoParserPluginContext
MathainoParserPluginLoader
MathainoParserPluginRegistry
Serializable
MathainoParserPluginRegistryNode
AbstractPlugin
MathainoCreatorPlugin
MathainoPlanNavigatorPlugin
AbstractParserPlugin
MathainoTemplateCreatorPlugin
MathainoAnalyzerCreatorPlugin
MathainoFixedCoordinateCreatorPlugin
MathainoTemplateParserPlugin
MathainoFixedCoordinateParserPlugin
MathainoPluginContext
MathainoCreatorPluginContext
MathainoParserPluginContext
MathainoPluginHandler
MathainoCreatorPluginLoader
MathainoParserPluginLoader
MathainoPluginRegistry
MathainoParserPluginRegistry
MathainoCreatorPluginRegistry
Serializable
MathainoPluginRegistryNode
Serializable
MathainoParserPluginRegistryNode
Serializable
MathainoCreatorPluginRegistryNode

(a) The Original Plugin Architecture

(b) The Refactored Plugin Architecture

Figure 5.2: Extracting a Common Superclass

one for the *parser* (i.e., the run-time components) plugins. This design also required two separate plugin loaders and two separate plugin registries. Figure 5.2(a) illustrates part of the class hierarchy as it stood in build nineteen.

We realized that this design was not extendable to handle new types of information we planed to include in the interaction model. Each new type of information to be included in the model would result in the construction of two new plugins, one in the *creator* branch and another in the *parser* branch. Synchronizing the two components and maintaining their correspondences would thus result in parallel modifications in the two branches.

Furthermore, if new plugins, not belonging in either of these two categories, were to be created, then another class hierarchy would have to be created and another plugin registry database schema would have to be implemented from scratch.

Quite beyond the limitations of the design's extensibility, there were also certain shortcomings at the code level. A lot of code was duplicated, as essentially many classes in this scheme (the initial abstract base classes for creator and parser plugins, the classes for maintaining the registry and the two plugin loaders) had overlapping functions.

For these reasons, a decision was made to extract a common plugin super class from these two separate plugin class trees. The resulting class diagram, as refactored in build twenty, is shown in Figure 5.2(b). The new design had several advantages over the old one.

1. In the new class hierarchy, the two separate plugin trees were combined into a single unified plugin tree. The class `MathainoPlugin` now formed the superclass for all Mathaino plugins, and thus, every plugin irrespective of whether it is a creator or parser plugin could now be treated simply as a `MathainoPlugin` if needed.

2. Also the plugin registry classes were combined into a unified hierarchy. Physically this enabled us to store the entire plugin registry in a single physical disk file leading to better registry database management.

3. Although the separate plugin loaders had to be retained for compatibility reasons, the `MathainoPluginHandler` class could now load any and all Mathaino plugins.

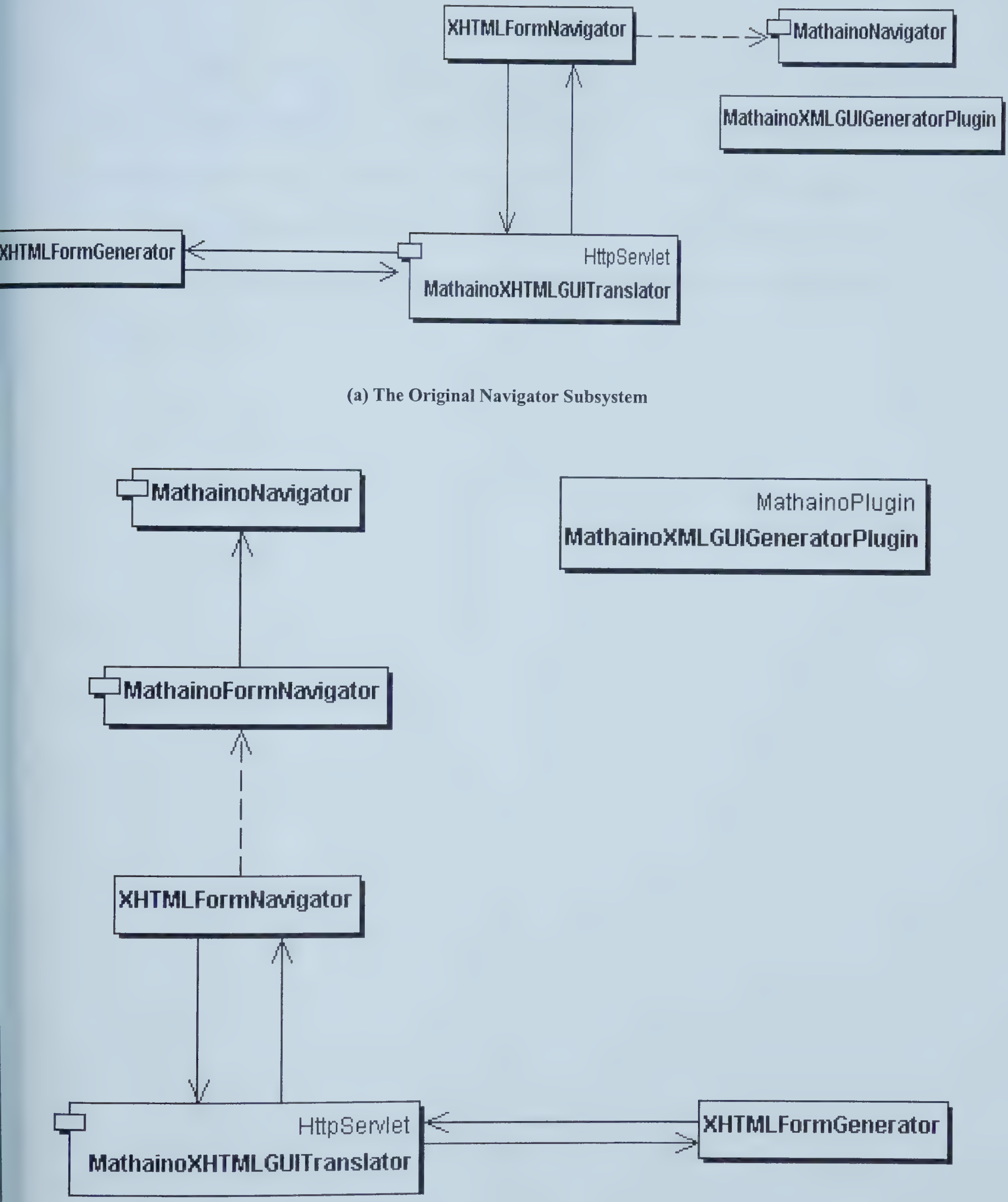
4. All the duplicated code was moved into the superclass reducing about 2000 lines of code.

5. The architecture was much more maintainable now as new plugins categories could now be introduced without introducing a separate class hierarchy tree for them.

5.4.2 Extracting the `FormNavigator` Class

Another substantial refactoring took place between builds forty-three and forty-four. By the time version forty-two passed acceptance testing, we had decided that one of the major objectives of Mathaino was to generate abstract, platform independent GUIs which could be implemented on a variety of platforms. Our chosen method for accomplishing this objective was to develop a suite of runtime components capable of interpreting the interaction model constructed at design time on different platforms utilizing the native widget sets of these platforms.

The forty-third build of Mathaino contained only one such runtime component, namely the XHTML model interpreter, Figure 5.3(a). While the `MathainoNavigator` class (which forms the core of the legacy screen navigator component shown in Figure 4.1) provided a higher layer of abstraction over the vanilla terminal emulator API, it was not sufficiently abstract. Any new runtime component interpreting the interaction model on a new platform would have to also navigate through a set of abstract forms, similar in nature to the XHTML forms navigated by the `XHTMLFormNavigator` yet different. Therefore, each of these new components would have to replicate some of the functionality of this class. Thus, it became clear that an independent navigation API for such abstract forms was needed, otherwise each GUI translator would have to implement its own form navigator. Since the `XHTMLNavigator` component already had such a form navigator it was decided to use the *Extract Class* refactoring [FOWLER99] to create this form navigator component.



(a) The Original Navigator Subsystem

(b) The Refactored Navigator Subsystem

Figure 5.3: Extracting the FormNavigator Class

Thus, build forty-four contains a separate form navigator component, `MathainoFormNavigator` in Figure 5.3(b), as a higher level navigation API for abstract GUI translators. While this scheme did not immediately lead to any significant code reduction, it helped tremendously in the long run as the new GUI translators did not require a separate platform dependent form navigator, making them much thinner and easier to implement.

Build	Packages	Classes	Public	Inner	LOC	Statements	Methods	Public	Variables	Public
19	10	63	63	63	7014	4249	711	502	665	23
20	10	68	68	62	7042	4283	714	506	657	23
43	14	101	101	119	12672	7951	1231	804	1180	52
44	14	103	103	119	12809	8046	1251	820	1194	52
87	22	137	137	169	18790	11735	1704	1073	1791	87

(a) Project Metrics

Build	LOC	Statements	LCOM	#Methods	Collaborators	Public	Public
						Methods	Variables
19	111.3	67.4	0.7	11.3	9.4	8	0.4
20	103.6	63	0.6	10.5	9.0	7.4	0.3
43	125.5	78.7	0.6	12.2	9.4	8	0.5
44	124.4	78.1	0.6	12.1	9.3	8	0.5
87	137.2	85.7	0.7	12.4	10.2	7.8	0.6

(b) Class Statistics (Averages)

Build	LOC	Statements	Cyclomatic Complexity	Collaborators
19	8.4	5.7	2.3	3.0
20	8.4	5.8	2.3	3.0
43	8.8	6.2	2.5	3.1
44	8.7	6.2	2.5	3.1
87	9.4	6.6	2.6	3.2

(c) Method Statistics (Averages)

Table 5.2: Summary Metrics of Mathaino's major builds (collected using JMetric)

5.4.3 Metrics on Mathaino's refactorings

Table 5.2 reports some summary metrics of the Mathaino code at five different milestones in the development process, collected with the JMetric tool [JMETRIC].

5.5 A Look At the Metrics

Qualitatively, we expected refactorings to improve design - in fact we employed refactorings primarily for this exact purpose. We have used JMetric to quantitatively measure the versions of Mathaino discussed in the subsections above so that we can better understand the impact of refactorings on the system design and code. More specifically, we were interested to see whether we could find quantitative evidence for Lehman's laws of evolution in this style of development also. Here are some observations.

1. Refactorings decrease the average LOC and the average number of statements of individual system classes.

2. Refactorings decrease the average number of methods of individual system classes.

3. Refactorings decrease the average number of collaborators of individual system classes.

4. For each subsequent version, irrespective of whether or not it is an extension or a refactoring of the previous one, all the above metrics, when applied at the project level, increase. The addition of new features explains the increase in these metrics for extensions. As each refactoring improves the quality of the code base it leads to an increase in the development speed. This explains the observation that all these metrics increase even in case of refactorings.

The *number of collaborators* metric measures coupling between classes. The higher the average number of collaborators for the system classes, the more dependent each class is on other classes to deliver its services. Metrics such as LOC and statements measure size. Thus, observations 1, 2, and 3 support the initial inference that *refactorings of the type extract common superclass or extract subclass decrease the average local complexity of the system design.*

Viewing refactorings as a form of *preventive maintenance*, the above statement conforms to Lehman's second law. Indeed, employing refactorings brings down the system complexity in this style of development lifecycle.

In the versions we measured we didn't see any significant impact on the method-related metrics, however, we believe that this may be due to the object oriented nature of refactorings performed in these versions. Further investigation might identify trends in the evolution of method-related metrics.

5.6 Some Conclusions

We have described in the two subsections above two major refactorings. Throughout the system evolution many other simpler refactorings have occurred, such as classes being moved across packages and methods being moved across classes in many cases. Our preliminary examination of this fairly small part of the data promises some interesting results.

Overall we have been quite satisfied with our experimentation with extreme programming and refactoring based development and would recommend it for other research projects in light of our current positive experiences with it.

Chapter 6

Conclusion

In Chapter 1 we had highlighted the various problems with the existing legacy system migration solutions. To recap, we had discussed the following problems with currently existing migration techniques:

1. Almost all white box migration techniques that have been proposed or tried out till now require a substantial human resource investment. In fact, it is estimated that a majority of programmers (four out of seven) are currently engaged in various white box migration or maintenance projects.

2. In addition to requiring a large investment in human resources, white box migration techniques are also cost and risk intensive.

3. Usually, non-terminally ill legacy systems do not require substantial reengineering and can be handled very effectively by black box migration techniques.

4. However, black box migration techniques like batch processing or object wrapping still require a non-trivial amount of manual intervention and, thus, are cost intensive.

5. Screen scrapping, which is an automated, cost effective and low risk black box migration technique suffers from a host of problems like ineffective user interface generation, limited protocol support, and limited platform support (only HTML).

Mathaino is a research prototype, which we have developed, primarily to solve some of these problems being faced by the legacy system migration community. Essentially, Mathaino is an intelligent, highly automated, low risk, black box migration solution for legacy information systems. The easiest way of classifying Mathaino is that it is an expert system shell for driving legacy information systems.

At the heart of Mathaino lie two core components known as the legacy trace analyzer and the legacy navigation planner (these were discussed in Chapter 3). Using the various intelligent learning algorithms that are coded into these two components, Mathaino can learn to navigate any given legacy system for performing pre-determined tasks on it, given a sufficient number of training examples.

The ability of Mathaino to learn to operate these legacy systems provides it with certain inherent advantages over previously proposed black box migration techniques. For example, once Mathaino has learnt to operate a given legacy system:

1. It can automatically infer the minimum set of information required from the user for performing a given task on it. Usually, this leads to a substantial reduction in the set of input information as Mathaino can automatically generate navigation commands and other redundant information for the given legacy system. As shown in Chapter 3 (see Table 3.1) the use of this technique significantly reduces the set of input information required for the execution of the PINE trace.

2. Mathaino also enables the user to impose a domain model on the observed user interface of the given legacy system consisting of objects relevant to that domain. Since Mathaino performs this domain modeling at the user interface level the process is low risk and substantially cost effective when compared to current object wrapping solutions. For an example of a Mathaino domain model for an email domain please see, Chapter 4, Section 4.4.

3. Unlike screen scrapping solutions, Mathaino can produce GUIs that are:

    a. Faithfully represent the reduced set of information required to carrying out a given task on the given the legacy system.
    b. Conform to the native platform look and feel and utilize the native widget set.
    c. Are much more user friendly than the original text based user interface.

   d. Can be used to drive a much wider variety of legacy systems. For example, Mathaino can successfully migrate legacy systems that operate using the VT100, HAL-API, and TCP-3270 protocols.

4. Also due to its abstraction oriented migration process, Mathaino can simultaneously migrate a given legacy system to a spectrum of platforms ranging from XHTML to WML to Java without any additional effort on the developer's part. Thus, the user can connect to the legacy systems from a multitude of platforms; each offering him a GUI specially customized for his current platform. Figures 4.1 and 4.4 highlight these multi-platform capabilities of Mathaino for the PINE trace.

5. Mathaino also includes an object oriented XML driver that can be used to drive the legacy systems that have been wrapped using Mathaino. By utilizing this XML driver, Mathaino can be used for integrating various legacy systems once appropriate domain objects have been created for their domain of operation. For more details on the Mathaino's XML driver please see Section 4.3.

6.1 Contributions

Thus, while building Mathaino we have tried to identify and provide solutions for the problems that plague the current migration solutions for non-terminally ill legacy systems. With Mathaino our prime areas of concentration have been to provide a legacy information system migration solution that:

1. Doesn't require a team of expert programmers to implement.

2. Does not require modifications to the existing legacy system.

3. Simultaneously migrates an existing legacy system to a multitude of platforms without requiring any extra effort on the developer's part.

4. Operates at the middle two tiers of a four-tier client-server architecture.

5. Is very cost effective.

6. Offers substantial risk reduction over existing LIS migration solutions.

7. Can effectively migrate non-terminally ill legacy systems at the user interface level.

8. Avoids the problems associated with competing black box migration solutions like screen scrapping, batch processing, object wrapping etc. and offers substantial benefits over them.

As already discussed, Mathaino has been designed for migration of legacy systems which can easily lend themselves to black box migration techniques (i.e. the set of non-terminally ill legacy systems). While this solution is again not appropriate for terminally ill legacy systems, there is a large collection of legacy systems that can benefit from it.

Further, as Mathaino concentrates on providing XHTML, WML and XML wrappers for an existing legacy system, this solution should be particularly appealing to the developers who are in a quandary over migrating their existing legacy systems to the web, in a resource limited, cost effective and risk free manner. To the best of our knowledge, till now, a simultaneous, multi-platform migration solution for legacy systems has not been attempted before, in either the BlackBox or WhiteBox migration domains. This added feature definitely makes Mathaino unique.

6.2 Some Concluding Thoughts…

Legacy system migration has continued to retain its position as one of the hot issues in the software engineering community for a while now. Of course, being a multi-trillion dollar issue, it amply deserves all the attention it gets. Any advance in legacy system migration technologies has an immediate, direct commercial impact on the entire IT industry. Hence, it is an extremely important issue. Unfortunately till now, no completely satisfactory solution exists for this problem.

Over the years, researchers working in the LIS migration field have concentrated on two major themes: invasive legacy code migration or the non-invasive black box migration. Almost without an exception, it has been assumed that since code migration is a hard problem it absolutely requires some form of knowledge based migration tool while non-invasive migration techniques have not been deemed complicated enough to require this approach. Thus, the class of non-invasive migration solutions like batch processing, and

screen scrapping remain essentially dumb solutions while a wide array of intelligent solutions have been tried out (albeit unsuccessfully) for code migration techniques.

The problem of applying AI to code understanding is very akin to the problem already being faced in the Natural Language Processing (NLP) domain. While the number of reserved words in a given computer language are usually quite restricted and the syntax of an artificial computer language is unambiguously parsable, this still does not imply that an automated code understanding tool can comprehend the human intention behind a particular piece of code construct, just as an NLP program, that can recognize the complete set of words in the English dictionary and has a fairly advanced natural grammar parsing algorithms, can successfully parse a given English sentence but still not actually understand it. This is the prime reason why even if code understanding tools can recognize relatively high level abstract concepts like algorithms, they still can't provide any insight into why a particular algorithm was preferred over another or how does a particular piece of code relate to the whole system at an architectural level. Hence, while automated code understanding is a very interesting AI problem its real world impact has been minimal.

"Usually evolution is a better solution than revolution". With Mathaino we have not tried to buck the currently emerging trends in the domain of LIS migration solutions. We agree that to be successful an LIS migration tool needs to be intelligent. However, since it is clear that the code understanding problem has proved to be a much harder nut to crack than initially anticipated, we believe that we must reorient the direction of our efforts to

solve a relatively easier problem. We believe that with the current state of technology, an intelligent solution at the user interface level can achieve a higher rate of success than one at the code level. For example, a single action at the user interface level can trigger an avalanche of inter-connections at the code level. If we can somehow decipher the meaning of this action at the interface level, we needn't analyze all its implications at the code level. Thus, we believe its time we tried building more intelligent migration tools working at the user interface level. This is the essence of Mathaino.

Of course, Mathaino, still, has its shortcomings. For example, the class of terminally ill legacy systems does not even fall under its domain of applicability. However, there does exist a large set of legacy systems that can still benefit from such a solution. We can vouch for this fact, as the Celcorp Corporation [CELCORP], which is one of the commercial sponsors of this project, has been steadily making commercial headway in the LIS migration market utilizing techniques very similar to Mathaino for its LIS migration efforts.

Further, we have comprehensively tested our proposed solution on a range of legacy information systems. We have already discussed the Pine legacy email reader as a running example throughout this thesis. In addition to Pine, we have used Mathaino to migrate a legacy system (access to which was provided to us courtesy of Celcorp) belonging to a large North American insurance company. We have also successfully tested our solution on the publicly available Hollis [HOLLIS] library system (for these and more examples of legacy systems that have been successfully migrated using

Mathaino please refer to Appendix A). Thus, we have demonstrated that Mathaino does offer a fairly robust solution within its domain.

In conclusion, we believe that with Mathaino we have developed a fairly easy to use, and a low cost and risk technique of dealing with legacy information systems that certainly provides an interesting alternative to the previously proposed solutions in this domain.

References

[AGGRAWA99] Agrawal R., Bayardo R., and Srikant R., “Athena: Mining-based interactive management of text databases”, *Research Report RJ 10153*, IBM Almaden Research Center, San Jose, CA 95120, July 1999.

[AIKE94] Aiken, P., Muntz, A., and Richards, R., “DoD legacy systems: reverse engineering data requirements”, *Communications of the ACM*, Vol. 37, Issue 5, 1994, pp. 26-41.

[ARRA00] Arranga, E.C., “Fresh From Y2k, What’s Next For Cobol”, *IEEE Software*, Vol. 17, Issue 2, March-April 2000, pp. 16-20.

[BECK00] Beck, K., “Embracing Change with eXtreme Programming”, *IEEE Computer*, Vol. 32, Issue 10, October 1999, pp. 70-77.

[BECK94] Beck, K., and Johnson, R., “Patterns Generate Architectures”, *Proceedings of the European Conference on Object-oriented Programming (ECOOP)*, Springer-Verlag, July 1994, pp. 139-149.

[BECK99] Beck, K., *Extreme Programming Explained: Embrace Change*, Addison-Wesley Publishing Co., October 1999.

[BENN95] Bennett, K.H., “Legacy Systems: Coping With Success”, *IEEE Software*, Vol. 12, Issue 1, January 1995, pp. 19-23.

[BISB97] Bisbal, J., Lawless, D., Wu, B., Grimson, J., Wade, V., Richardson, and R., Sullivan, D., “A Survey of Research into Legacy System Migration”, Trinity College Dublin, External Technical Reports, *available from the world wide web: ftp://ftp.cs.tcd.ie/pub/tech-reports/reports.97/TCD-CS-1997-01.ps.gz*, 1997.

[BISB99] Bisbal, J., Lawless, D., Wu, B., and Grimson, J., “Legacy information systems: issues and directions”, *IEEE Software*, Vol. 16, Issue 5, September-October 1999, pp. 103-111.

[BLACK98] Celcorp, *Blacksmith Apprentice: A Programmer Introduction*, Celcorp, 1998.

[BOEHM88] Boehm, B., “A spiral model of software development and enhancement”, *IEEE Computer*, Vol. 21, Issue 5, May 1988, pp. 61-72.

[BRAY95] Bray, O. and Hess, M.M., “Reengineering a configuration-management system”, *IEEE Software*, Vol. 12, Issue 1, January 1995, pp. 55-63.

[CARR00] Carr, D., and Kizior, R.J., “The case for continued Cobol education”, *IEEE Software*, Vol. 17, Issue 2, March-April 2000, pp. 33-36.

[CARR98] Carr, D. F., “Web-Enabling Legacy Data When Resources Are Tight”, *Internet World*, August 10, 1998.

[CELCORP] Celcorp, *http://www.celcorp.com.*

[CHIK90] Chikofsky, E.J., and Cross, J.H., “Reverse engineering and design recovery: a taxonomy”, *IEEE Software*, Vol. 7, Issue 1, January 1990, pp. 13-17.

[COME01] Comella-Dorda, S., “Black-box Modernization of Information Systems”, *available from the world wide web: http://www.sei.cmu.edu/str/descriptions/blackbox.html*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, March 2001.

[DELUCIA98] DeLucia, A., and Cimitile, A., “Incremental Migration Strategies: Data Flow Analysis For Wrapping”, *Proceedings of the Working Conference on Reverse Engineering (WCRE’98)*, IEEE Press, Honolulu, Hawaii, USA, October 1998, pp. 59-68.

[DERI97] Deri, L., “Yasmin: A Component Based Architecture for Software Applications”, *Proceedings of the 8th International Conference of Software Technology and Engineering Practice (STEP’97)*, IEEE Press, 1997.

[DYNABOOK] Williams, L., et. al., “What Is eXtreme Programming (sidebar)”, *IEEE Software*, Vol. 17, Issue 4, July-August 2000, pp. 19-25.

[EISENSTEI00] Eisenstein J., and Puerta A., “Adaptation in Automated User-Interface Design”, *Proceedings of Intelligent User Interfaces 2000*, ACM Press, New Orleans, LA, 9-12 January 2000, pp. 74-81.

[ENG96] Eng, E., “QT GUI Toolkit: Porting graphics to multiple platforms using a GUI toolkit”, *Linux Journal,* ACM Press, Vol. 1996, Issue 31es, Article No. 2, 1996.

[FOLD96] Howe, D., “The Free On-line Dictionary of Computing”, *available from the world wide web: http://wombat.doc.ic.ac.uk.*

[FOURLAYER] Portland Pattern Repository, “Four Layer Architecture”, *available from the world wide web: http://c2.com/cgibin/wiki?FourLayerArchitecture*, Portland Pattern Repository.

[FOWLER99] Fowler, M., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Publishing Co., August 1999.

[FRASER00] Fraser, S. et. al., “Hacker or Hero? – Extreme Programming Today (panel session)”, *Conference on Object-Oriented Programming, Systems, Languages, and Applications on Addendum to the 2000 proceedings (OOPSLA 2000)*, ACM Press, 2000, pp. 5-7.

[FUJITSU01] Fujitsu, “Cobol for Microsoft.NET Framework”, *available from the world wide web: http://www.adtools.com/info/whitepaper/dotnet_whitepaper.html*, May 2001,

[GAMMA93] Gamma, E., Helm, R., Johnson, R., and Vlissides J., "Design Patterns: Abstraction and Reuse of Object Oriented Design", *Proceedings of the European Conference on Object-oriented Programming (ECOOP)*, Springer-Verlag, July 1993, pp. 406-431.

[GLASS98] Glass, R.L., "Maintenance: less is not more", *IEEE Software*, Vol. 15, Issue 4, July-August 1998, pp. 67-68.

[GOLD98] Gold, N., "The Meaning of Legacy Systems", *available from the world wide web: http://www.dur.ac.uk/~dcs0elb/tr_07-98.ps*, January 1998.

[HILL01] Hill, T., "Merant: Egility Edge", *available from the world wide web: http://www.gartner.com/webletter/merant*, March 2001.

[HOLLIS] The Harvard University Library Union Catalog, *http://hollis.harvard.edu.*

[JAVAS00] Sun Microsystems, "Java™ Servlet Specification, Version 2.3", *available from the world wide web: http://java.sun.com/aboutJava/communityprocess/first/jsr053/servlet23_PDF.pdf*, October 20, 2000.

[JMETRIC] JMetric Tool, *available from the world wide web: http://www.jmetric.com.*

[KARLSSON00] Karlsson, E.A., Andersson, L.G. and Leion, P., "Daily Build and feature development in large distributed projects", *Proceedings of the 22nd international conference on Software engineering (ICSE 2000)*, ACM Press, Limerick, Ireland, June 4-11, 2000, pp. 649-658.

[KONGSTR99] Kong, L., Stroulia E., and Matichuk B, "Legacy Interface Migration: A Task-Centered Approach", *Proceedings of the 8th International Conference on Human-Computer Interaction*, Lawrence Erlbaum Associates, Munich, Germany, August 22-27, 1999, 1167–1171.

[LEHMAN00] Lehman, M.M., Perry, D.E., Ramil, J.F., Turski, W.M., and Wernick, P., "Metrics and laws of software evolution – the nineties view". *Proceedings of the 4th International Symposium on Software Metrics,* IEEE Press, Albuquerque, New Mexico, 2000.

[LEHMAN98] Lehman, M. M., "Software's future: Managing evolution", *IEEE Software*, Vol. 15, Issue 1, January-February 1998, pp. 40-44.

[MARTIN00] Martin, R.C., "eXtreme Programming Development through Dialog", *IEEE Software*, Vol. 17, Issue 4, July-August 2000, pp. 12-13.

[MCDERM98] McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., and Wilkins, D. "The PDDL Planning Domain Definition Language", *The AIPS-98 Planning Competition Committee*, 1998

[MERLO95] Merlo, E., Girard, J., Gagne, P., Kontogiannis, K., Hendren L., Panangaden, P., and Mori, R., "Reengineering user interfaces", IEEE Software, Vol. 12, Issue 1, January 1995, pp. 64-73.

[MOORE94] Moore, M., Rugaber, S. and Seaver, P., "Knowledge Based User Interface Migration", *Proceedings of the 1994 International Conference on Software Maintenance (ICSM'94)*, IEEE Press, Victoria, British Columbia, Canada, September 1994.

[MOORE96] Moore, Melody. "Representation Issues for Reengineering Interactive Systems", *ACM Computing Surveys*, Vol. 28, Issue 4es, Article No. 199, December 1996.

[MUNSON98] Munson, J.C., "Software lives too long", *IEEE Software*, Vol. 15, Issue 4, July-August 1998, pp. 18, 20.

[NEWKIRK00] Newkirk, J. and Martin, R.C., "Extreme Programming in Practice", *Conference on Object-Oriented Programming, Systems, Languages, and Applications on Addendum to the 2000 proceedings (OOPSLA 2000)*, ACM Press, 2000, pp. 25-26.

[NICHOLS91] Nicholson, R.T., "Designing a portable GUI toolkit", *Dr. Dobbs Journal,* ACM Press, Vol. 1991, Issue Jan (1991), Article No. 4, January 1991.

[PANGIA] Pangia, "Solving Integration Challenges with Middleware", Pangia Corporation, *available from the world wide web: http://www.pangia.com/pangiacorp/middle.html*.

[PERR95] Perrochon, L., "W3 Middleware: Notions and Concepts", *available from the world wide web: ftp://ftp.inf.ethz.ch/pub/publications/papers/is/ea/4www95.html*, 1995.

[PHNXGRP] Phoenix Group, "Legacy Systems Wrapping with Objects", *available from the world wide web: http://www.phxgrp.com/jodewp.htm*, August 1997.

[ROBINSON99] Robinson, M., and Vorobiev, P.A., *Swing*, Manning Publications, December 1999.

[ROYSE70] Royse, W., "Managing the development of large software systems", *Tutorial: Software Engineering Project Management, Computer Society of the IEEE*, Washington, D.C., 1970, pp. 118-127.

[RUGA98] Rugaber, S., and White, J., "Restoring a legacy: lessons learnt", *IEEE Software*, Vol. 15, Issue 4, July-August 1998, pp. 28-33.

[SADOSKI00] Sadoski, D. and Comella-Dorda, S., "", *available from the world wide web: http://www.sei.cmu.edu/str/descriptions/threetier_body.html*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, February 2000.

[SHAW95] Shaw, M., "Patterns for software architecture", In J. Coplien and D. Schmidt, editors, *Pattern Languages for Program Design (PLoP)*, Addison-Wesley Publishing Co., 1995, pp. 453-461.

[SHAW96] Shaw, M., and Garlan, D., *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, April 1996.

[SHEPHERD96] Shepherd, G. and Wingo, S., *MFC Internals*, Addison-Wesley Publishing Co., May 1996.

[SIDDIQI00] Siddiqi, J., "An Exposition of XP But No Position on XP", *IEEE Dynabook on Extreme Programming, available from the world wide web: http://computer.org/seweb/dynabook/index.htm*, IEEE Press, 2000.

[SNEED92] Sneed, H.M., "Migration of Procedurally Oriented Cobol Programs in an Object-Oriented Architecture", *Proceedings of the Conference on Software Maintenance 1992*, IEEE Press, Orlando, Florida, 1992, pp. 105-116.

[SNEED95] Sneed, H.M., "Planning the reengineering of legacy systems", *IEEE Software*, Vol. 12, Issue 1, January 1995, pp. 24-34.

[SPINELLIS00] Spinellis, D., "Taking Common Sense to the Extreme", *IEEE Software*, Vol. 17, Issue 4, July-August 2000, pp. 113-114.

[STROULIA00] Stroulia E., El-Ramly M., Sorenson P., and Penner R., "Legacy Systems Migration in CelLEST", *22nd International Conference on Software Engineering*, ACM Press, Limerick, Ireland, June 4-11, 2000, pp. 790.

[STROULIA01] Stroulia, E., and Leitch, R., "The space station operations control software: A case study in architecture maintenance", *In IEEE, editor, HICSS 2001, 34th Annual Hawaii International Conference on System Sciences*, 2001.

[STROULIA99] Stroulia E., El-Ramly M., Kong L., Sorenson P., and Matichuk B., "Reverse Engineering Legacy Interfaces: An Interaction-Driven Approach", *6th Working Conference on Reverse Engineering*, October 6-8, 1999, Atlanta, Georgia USA.

[VALID00] W3C, "HTML Validation Service", *available from the world wide web: http://validator.w3.org/*, June 30, 2000.

[WAPFOUR] The WAP Forum, *http://www.wapforum.org*.

[WEID97] Weiderman, N., Northrop, L., Smith, D., Tilley, S. and Wallnau, K., "Implications of Distributed Object Technology for Reengineering", *CMU/SEI-97TR -005/ESC-TR-97-005*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, June 1997.

[WEIZ83] Weizenbaum, J., "ELIZA - a computer program for the study of natural language communication between man and machine", *Communications of the ACM*, Volume 26, Issue 1, 1983, pp.23-28.

[WILLIAMS00] Williams, L., Kessler, R., Cunningham, W. and Jeffries, R., "Strengthening the Case for Pair Programming", *IEEE Software*, Vol. 17, Issue 4, July-August 2000, pp. 19-25.

[WMLDTD00] WAP Forum, "WML 1.1 DTD", *available from the world wide web: http://www.wapforum.org/DTD/wml_1.1.dtd*, 2000.

[WONG95] Wong, K., Tilley, R., Muller, H., and Storey M., "Structural redocumentation: a case study", *IEEE Software*, Vol. 12, Issue 1, January 1995, pp. 46-54.

[WU97] B., Wu, Lawless, D., Bisbal, J., Richardson, R., Grimson, J., Wade, V., and Sullivan, D., "The Butterfly Methodology: A Gateway-free Approach for Migrating Legacy Information", *Proceedings of the 3rd IEEE Conference on Engineering of Complex Computer Systems (ICECCS '97)*, September 1997, pp. 200-205.

[XHTML00] Pemberton, S. et. al., "XHTML™ 1.0: The Extensible HyperText Markup Language", *available from the world wide web: http://www.w3.org/TR/xhtml1/*, January 26, 2000.

[XML1.0] W3C, "Extensible Markup Language (XML) 1.0 (Second Edition)", W3C Recommendation, *available from the world wide web: http://www.w3.org/TR/2000/REC-xml-20001006*, October 2000.

[ZHANGSTR01] Zhang, H., and Stroulia E., "Babel: Application Integration through XML specification of Rules", *23rd International Conference on Software Engineering (ICSE 2001)*, May 12-19, Toronto, Canada, 2001.

Appendix A

Migrating the Hollis Library System

Introduction

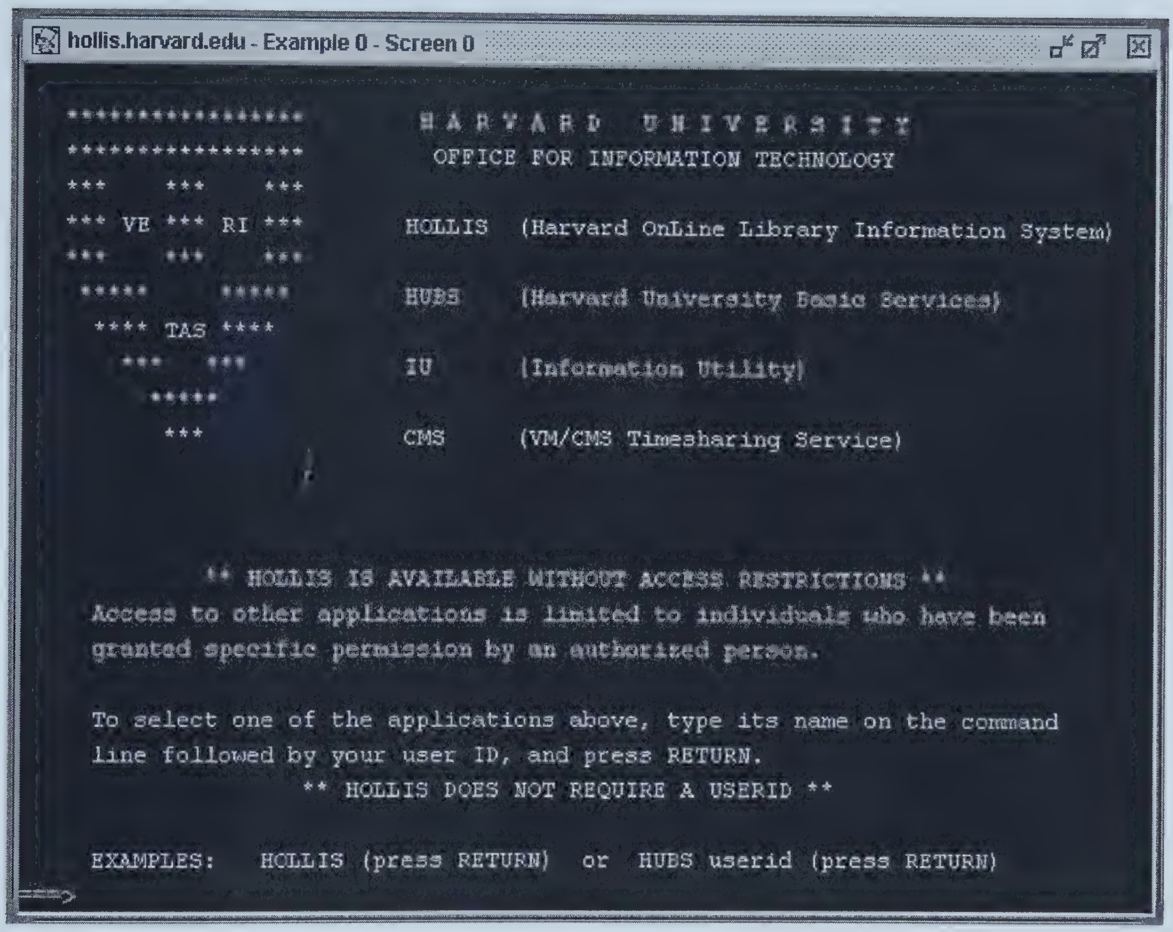
This appendix lists the original screens shots for the Hollis library system and the migrated interface constructed for it using Mathaino. Hollis can be accessed from hollis.harvard.edu. It uses TCP-3270 as its terminal emulation protocol.

The migrated GUI screen shots have been divided into three categories:

1. The Abstract GUI Forms.
2. The XHTML GUI Forms rendered using the XHTML translator.
3. The WML GUI Forms rendered using the WML translator.

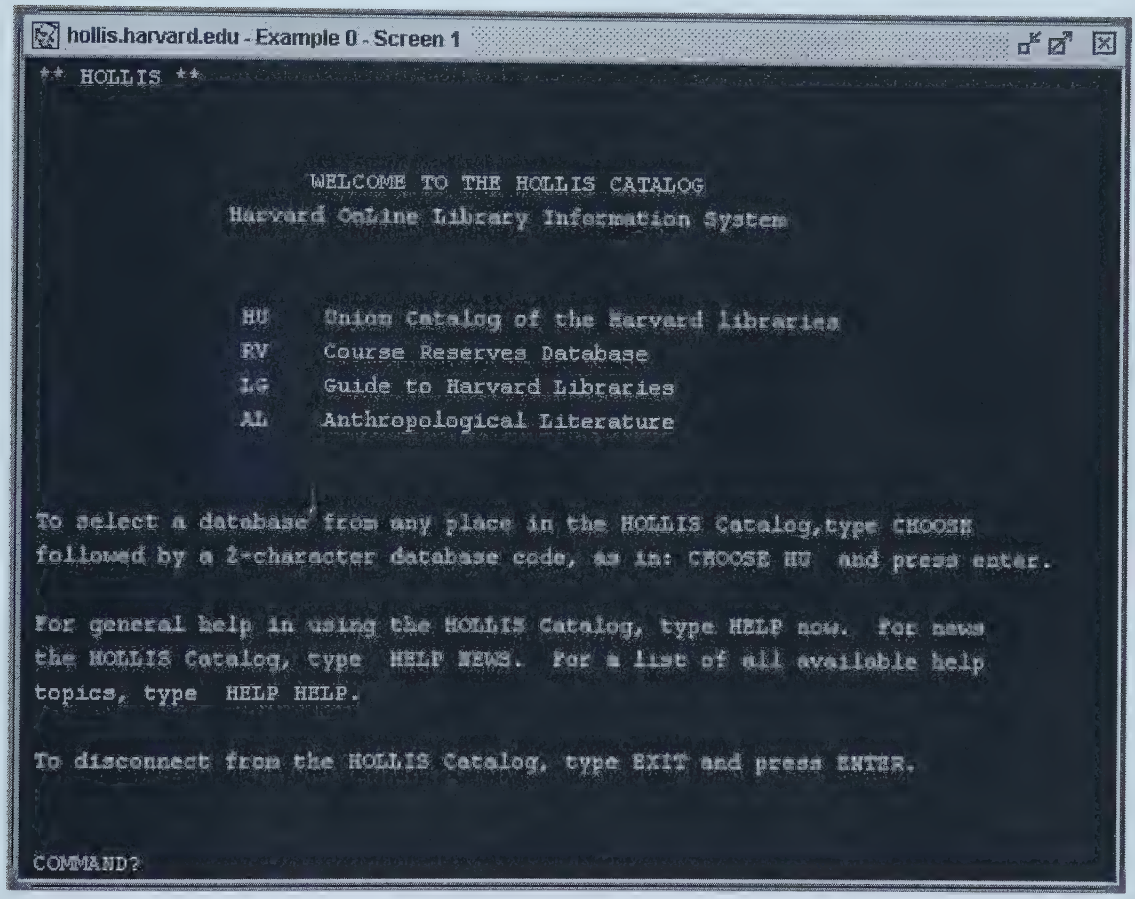
The following pages illustrate these screen shots.

Screen 1



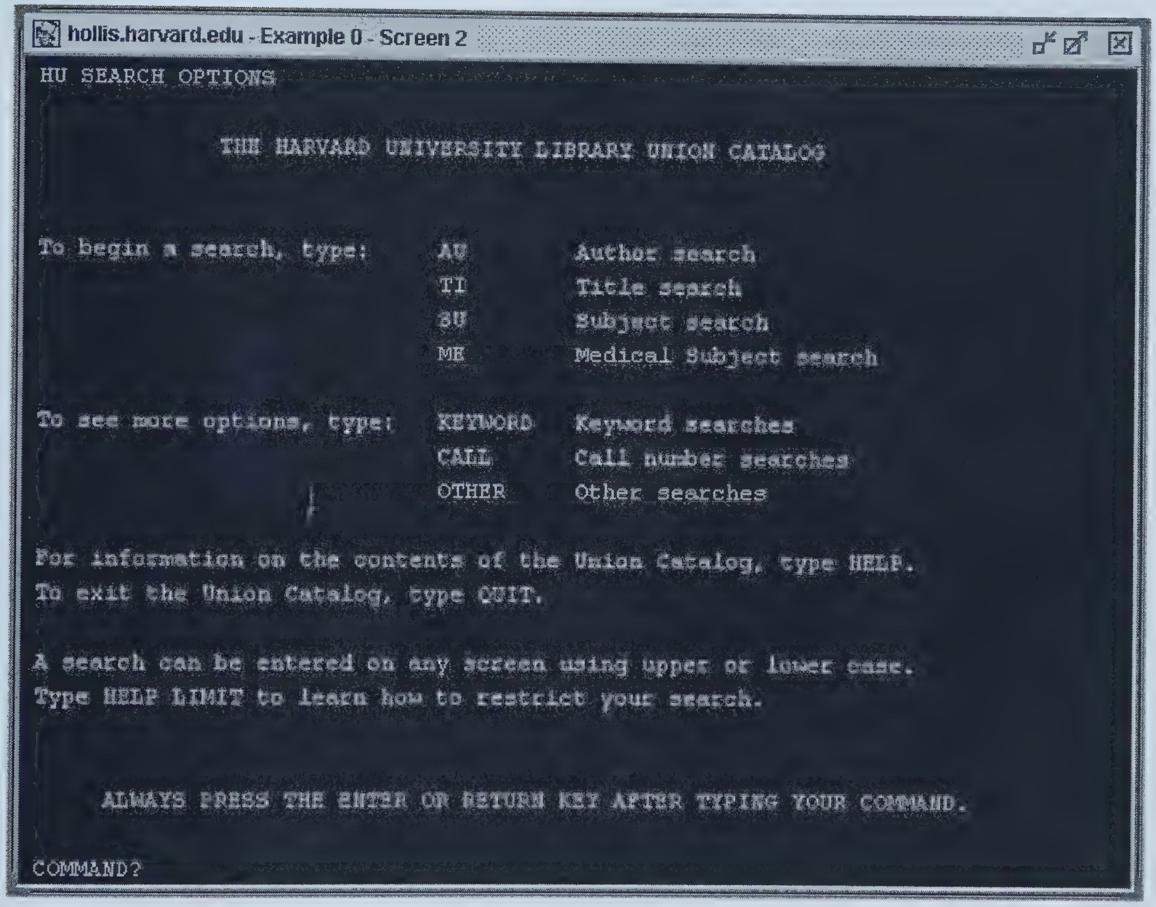
```
*****************        H A R V A R D   U N I V E R S I T Y
*****************         OFFICE FOR INFORMATION TECHNOLOGY
***    ***    ***
*** VE *** RI ***       HOLLIS  (Harvard OnLine Library Information System)
***    ***    ***
 *****     *****        HUBS    (Harvard University Basic Services)
  **** TAS ****
    ***   ***           IU      (Information Utility)
      *****
       ***              CMS     (VM/CMS Timesharing Service)



          ** HOLLIS IS AVAILABLE WITHOUT ACCESS RESTRICTIONS **
  Access to other applications is limited to individuals who have been
  granted specific permission by an authorized person.

  To select one of the applications above, type its name on the command
  line followed by your user ID, and press RETURN.
                 ** HOLLIS DOES NOT REQUIRE A USERID **

  EXAMPLES:   HOLLIS (press RETURN)  or  HUBS userid (press RETURN)
==>
```

Screen 2



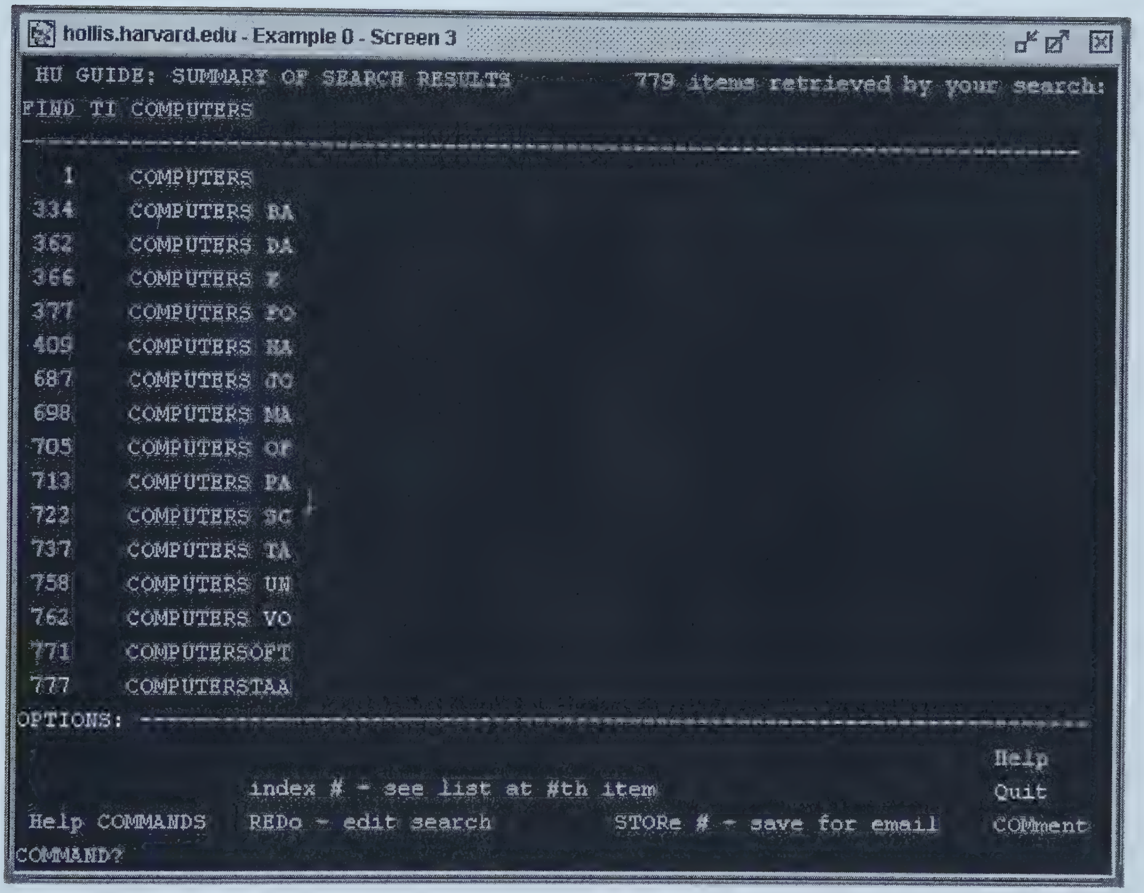
```
** HOLLIS **


                      WELCOME TO THE HOLLIS CATALOG
                 Harvard OnLine Library Information System


                  HU    Union Catalog of the Harvard libraries
                  RV    Course Reserves Database
                  LG    Guide to Harvard Libraries
                  AL    Anthropological Literature


To select a database from any place in the HOLLIS Catalog,type CHOOSE
followed by a 2-character database code, as in: CHOOSE HU  and press enter.

For general help in using the HOLLIS Catalog, type HELP now.  For news
the HOLLIS Catalog, type  HELP NEWS.  For a list of all available help
topics, type  HELP HELP.

To disconnect from the HOLLIS Catalog, type EXIT and press ENTER.


COMMAND?
```

Screen 3



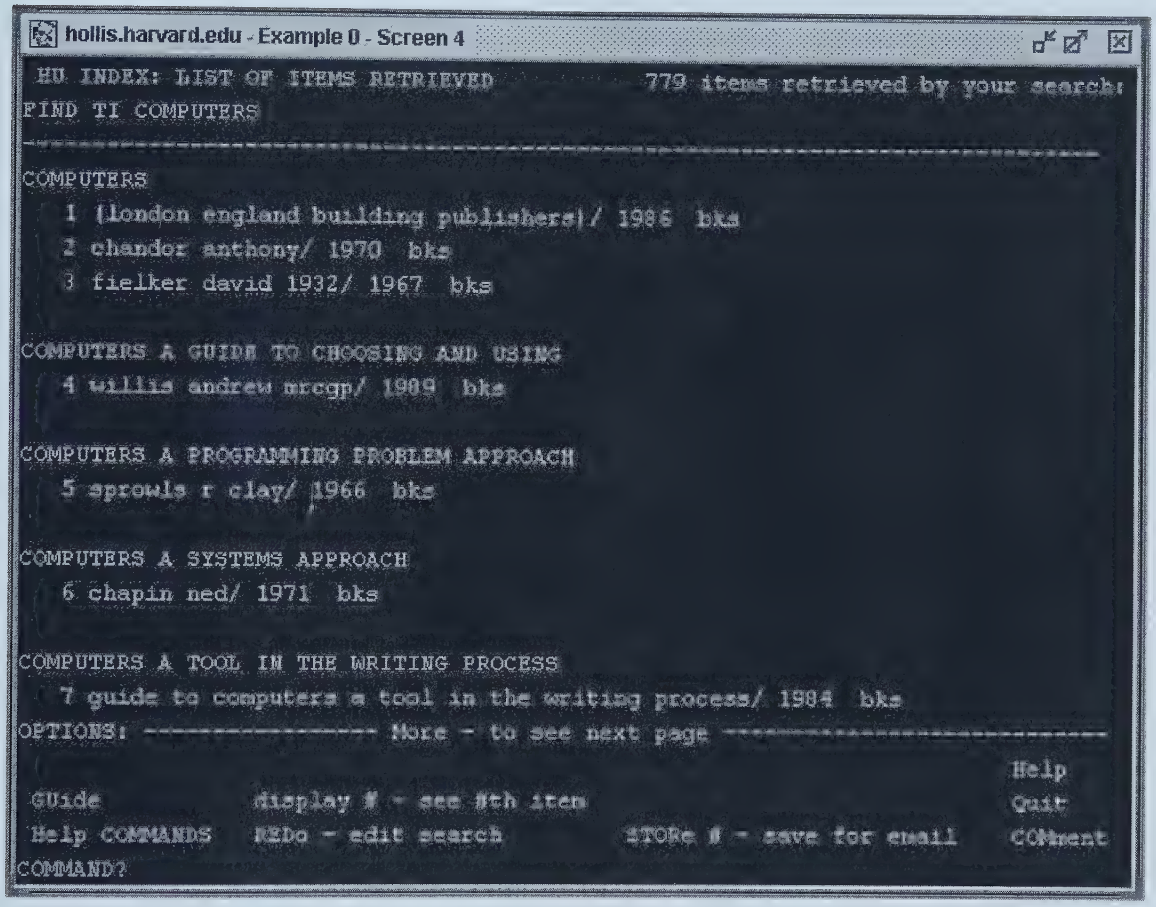
```
HU SEARCH OPTIONS

           THE HARVARD UNIVERSITY LIBRARY UNION CATALOG


To begin a search, type:     AU         Author search
                             TI         Title search
                             SU         Subject search
                             ME         Medical Subject search

To see more options, type:   KEYWORD    Keyword searches
                             CALL       Call number searches
                             OTHER      Other searches

For information on the contents of the Union Catalog, type HELP.
To exit the Union Catalog, type QUIT.

A search can be entered on any screen using upper or lower case.
Type HELP LIMIT to learn how to restrict your search.


   ALWAYS PRESS THE ENTER OR RETURN KEY AFTER TYPING YOUR COMMAND.

COMMAND?
```

Screen 4



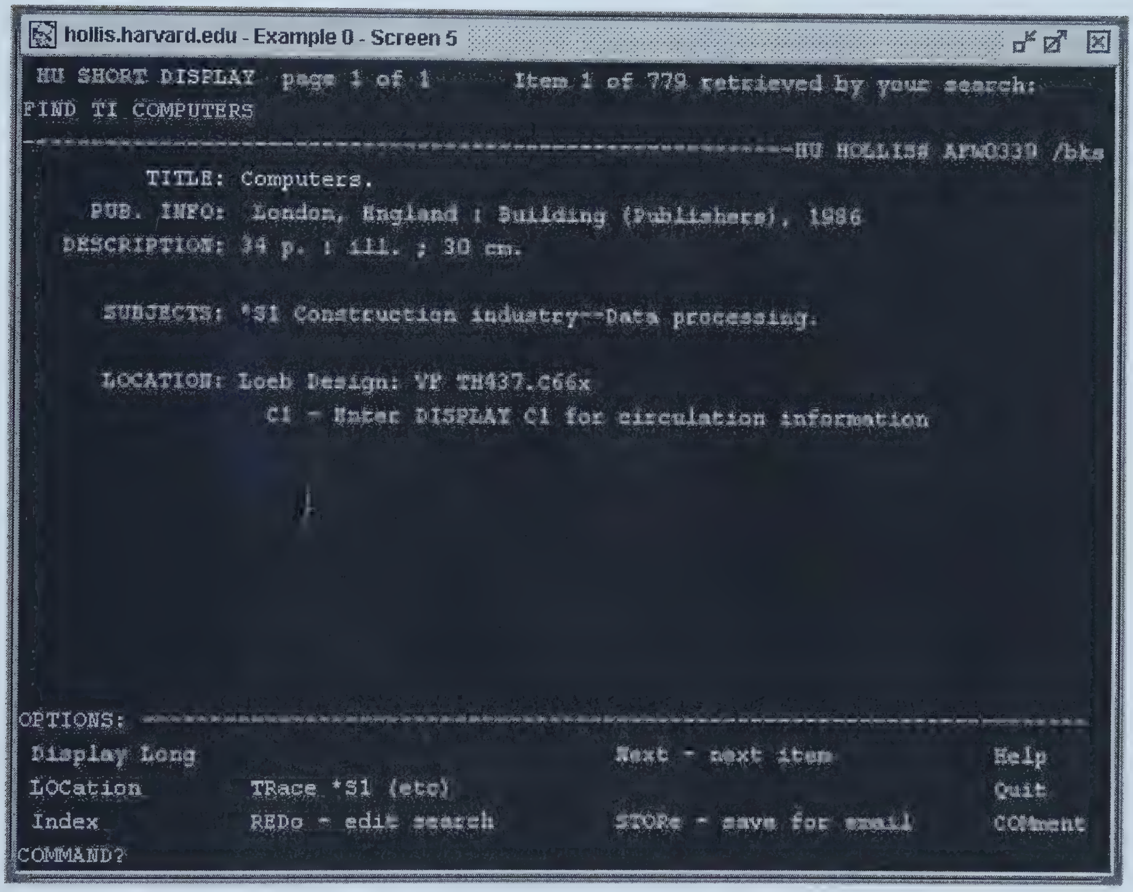
```
 HU GUIDE: SUMMARY OF SEARCH RESULTS         779 items retrieved by your search:
FIND TI COMPUTERS
-------------------------------------------------------------------------------
   1     COMPUTERS
 334     COMPUTERS BA
 362     COMPUTERS DA
 366     COMPUTERS F
 377     COMPUTERS FO
 409     COMPUTERS HA
 687     COMPUTERS JO
 698     COMPUTERS MA
 705     COMPUTERS OF
 713     COMPUTERS PA
 722     COMPUTERS SC
 737     COMPUTERS TA
 758     COMPUTERS UN
 762     COMPUTERS VO
 771     COMPUTERSOFT
 777     COMPUTERSTAA
OPTIONS: ----------------------------------------------------------------------
                                                                     Help
                index # - see list at #th item                       Quit
 Help COMMANDS  REDo - edit search        STORe # - save for email   COMment
COMMAND?
```

Screen 5



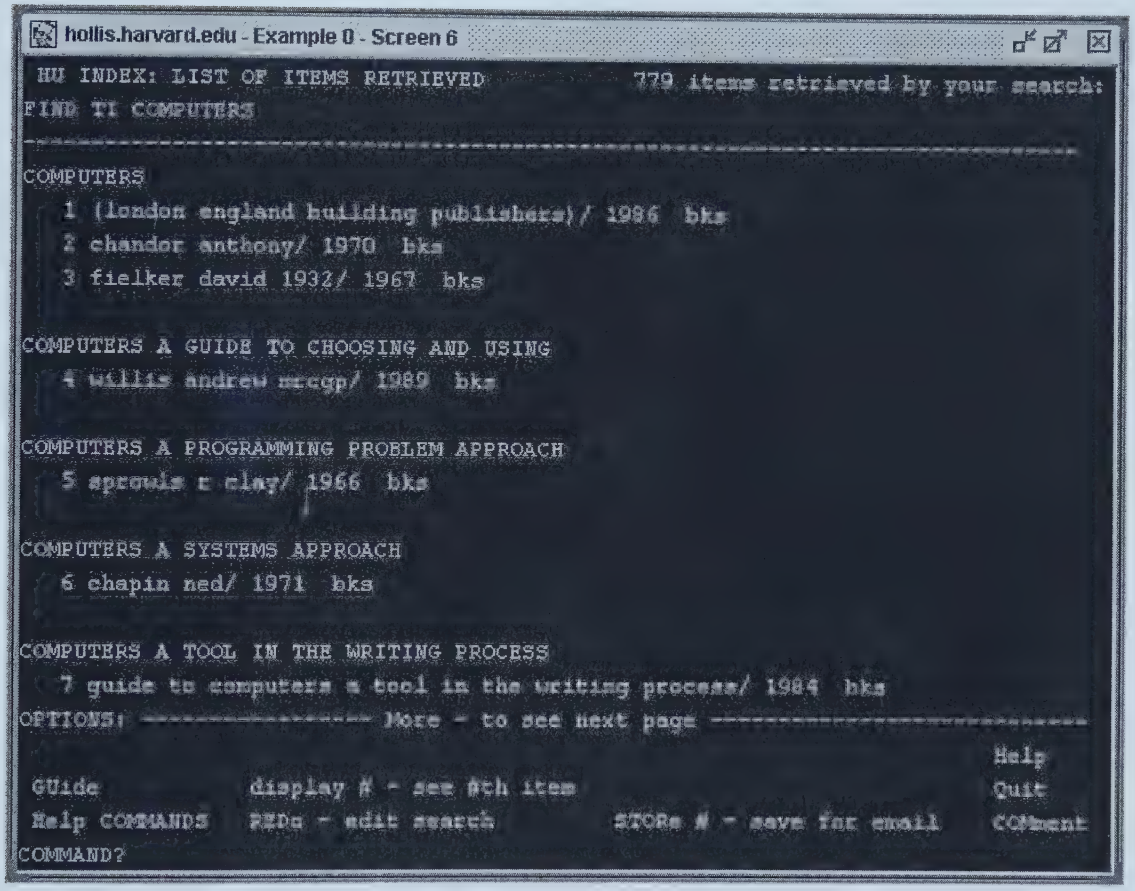
```
 HU INDEX: LIST OF ITEMS RETRIEVED           779 items retrieved by your search:
FIND TI COMPUTERS
---------------------------------------------------------------------------------
COMPUTERS
   1 [london england building publishers]/ 1986  bks
   2 chandor anthony/ 1970  bks
   3 fielker david 1932/ 1967  bks

COMPUTERS A GUIDE TO CHOOSING AND USING
   4 willis andrew mrcgp/ 1989  bks

COMPUTERS A PROGRAMMING PROBLEM APPROACH
   5 sprowls r clay/ 1966  bks

COMPUTERS A SYSTEMS APPROACH
   6 chapin ned/ 1971  bks

COMPUTERS A TOOL IN THE WRITING PROCESS
   7 guide to computers a tool in the writing process/ 1984  bks
OPTIONS: ---------------- More - to see next page ---------------------------
                                                                          Help
 GUide            display # - see #th item                                Quit
 Help COMMANDS    REDo - edit search          STORe # - save for email    COMment
COMMAND?
```

Screen 6



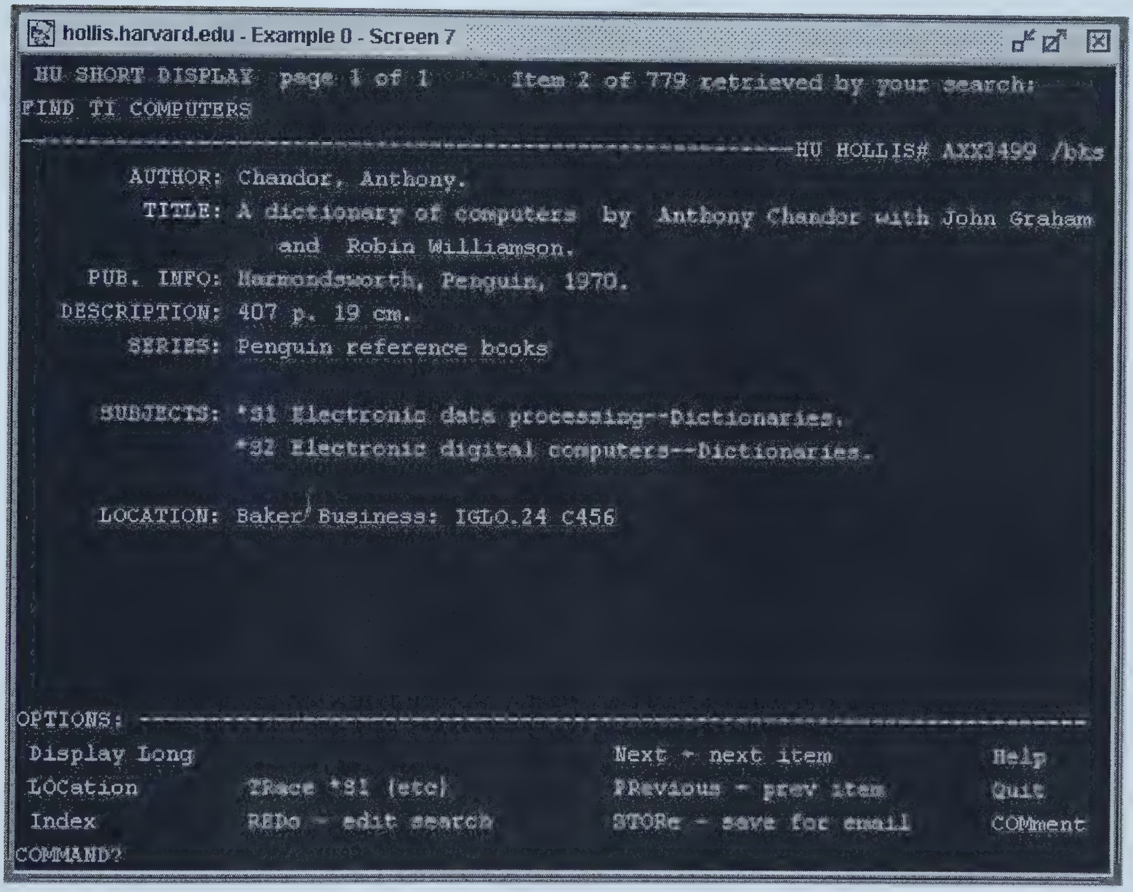
```
 HU SHORT DISPLAY  page 1 of 1       Item 1 of 779 retrieved by your search:
FIND TI COMPUTERS
---------------------------------------------------------HU HOLLIS# AFW0339 /bks
        TITLE: Computers.
    PUB. INFO:  London, England : Building (Publishers), 1986
  DESCRIPTION: 34 p. : ill. ; 30 cm.

     SUBJECTS: *S1 Construction industry--Data processing.

     LOCATION: Loeb Design: VF TH437.C66x
                 C1 - Enter DISPLAY C1 for circulation information









OPTIONS: ------------------------------------------------------------------------
 Display Long                               Next - next item            Help
 LOCation         TRace *S1 (etc)                                       Quit
 Index            REDo - edit search        STORe - save for email      COMment
COMMAND?
```

Screen 7



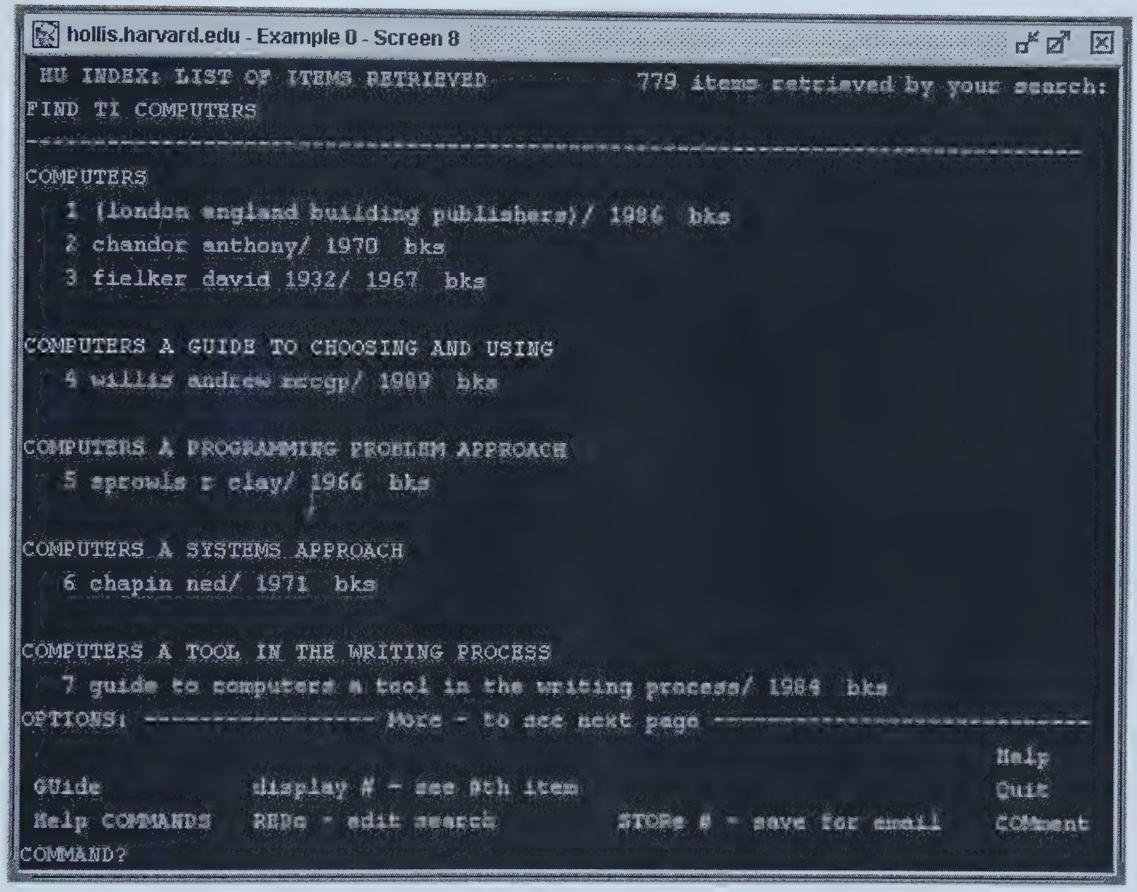
```
 HU INDEX: LIST OF ITEMS RETRIEVED            779 items retrieved by your search:
FIND TI COMPUTERS
-------------------------------------------------------------------------------
COMPUTERS
   1 (london england building publishers)/ 1986  bks
   2 chandor anthony/ 1970  bks
   3 fielker david 1932/ 1967  bks

COMPUTERS A GUIDE TO CHOOSING AND USING
   4 willis andrew mrcgp/ 1989  bks

COMPUTERS A PROGRAMMING PROBLEM APPROACH
   5 sprowls r clay/ 1966  bks

COMPUTERS A SYSTEMS APPROACH
   6 chapin ned/ 1971  bks

COMPUTERS A TOOL IN THE WRITING PROCESS
   7 guide to computers a tool in the writing process/ 1984  bks
OPTIONS: ----------------- More - to see next page --------------------------
                                                                    Help
 GUide            display # - see #th item                          Quit
 Help COMMANDS    REDo - edit search          STORe # - save for email   COMment
COMMAND?
```

Screen 8



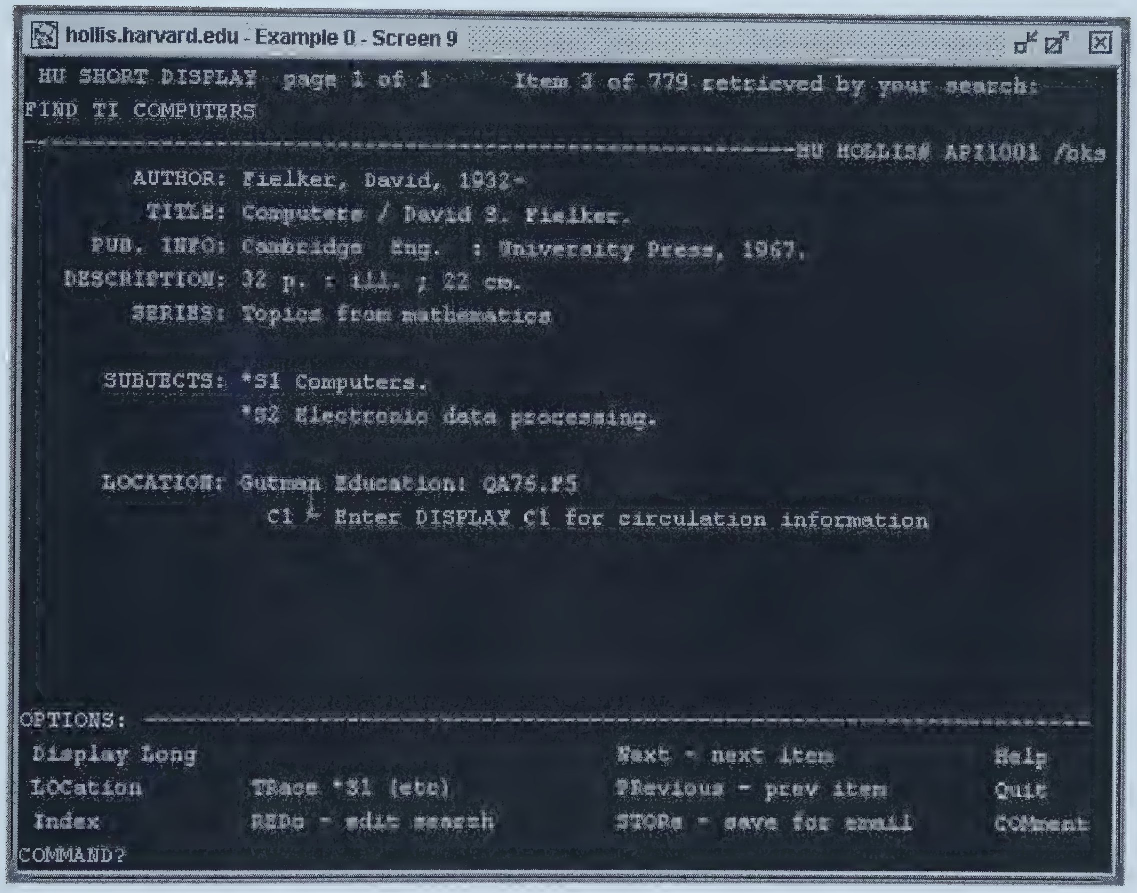
```
 HU SHORT DISPLAY  page 1 of 1      Item 2 of 779 retrieved by your search:
FIND TI COMPUTERS
------------------------------------------------------HU HOLLIS# AXX3499 /bks
       AUTHOR: Chandor, Anthony.
        TITLE: A dictionary of computers  by  Anthony Chandor with John Graham
                 and  Robin Williamson.
    PUB. INFO: Harmondsworth, Penguin, 1970.
  DESCRIPTION: 407 p. 19 cm.
       SERIES: Penguin reference books

     SUBJECTS: *S1 Electronic data processing--Dictionaries.
               *S2 Electronic digital computers--Dictionaries.

     LOCATION: Baker Business: IGLO.24 C456




OPTIONS: ---------------------------------------------------------------------
 Display Long                               Next - next item            Help
 LOCation        TRace *S1 (etc)            PRevious - prev item        Quit
 Index           REDo - edit search         STORe - save for email      COMment
COMMAND?
```

Screen 9



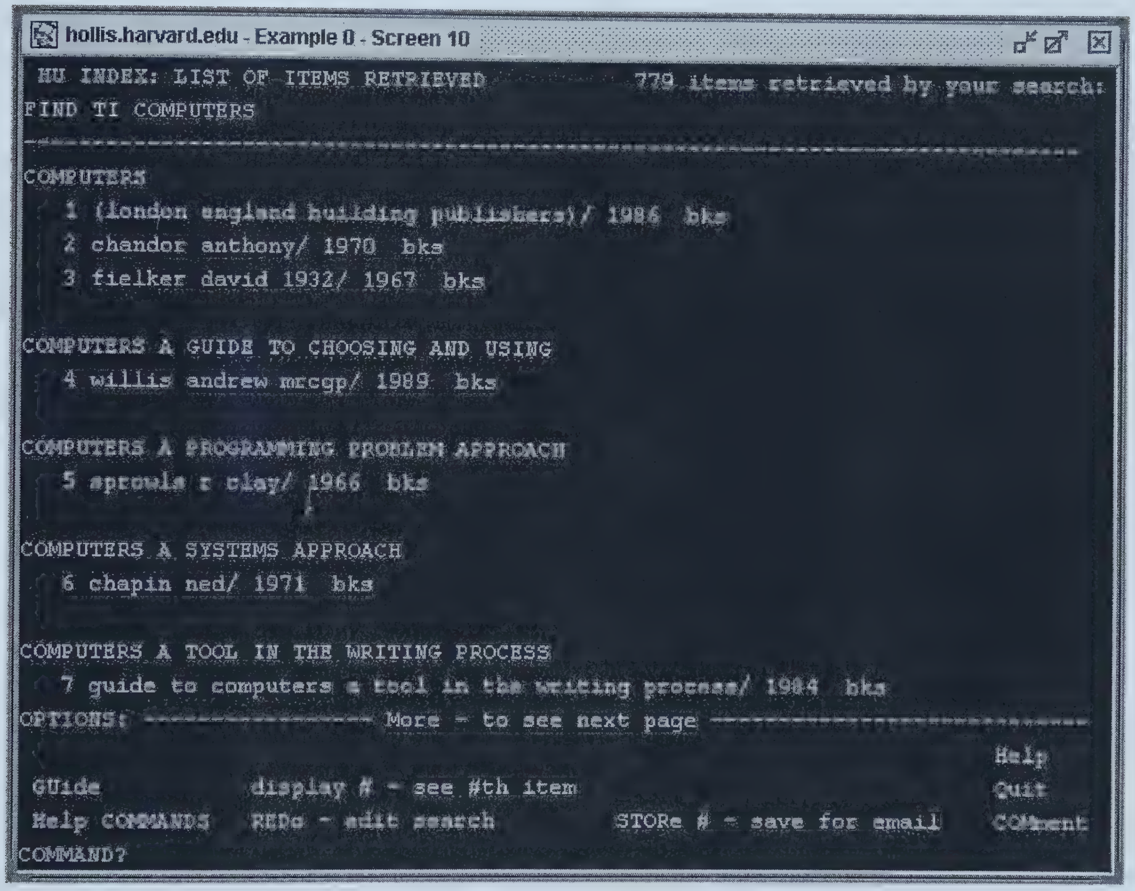
```
 HU INDEX: LIST OF ITEMS RETRIEVED          779 items retrieved by your search:
FIND TI COMPUTERS
--------------------------------------------------------------------------------
COMPUTERS
   1 (london england building publishers)/ 1986  bks
   2 chandor anthony/ 1970  bks
   3 fielker david 1932/ 1967  bks

COMPUTERS A GUIDE TO CHOOSING AND USING
   4 willis andrew mrcgp/ 1989  bks

COMPUTERS A PROGRAMMING PROBLEM APPROACH
   5 sprowls r clay/ 1966  bks

COMPUTERS A SYSTEMS APPROACH
   6 chapin ned/ 1971  bks

COMPUTERS A TOOL IN THE WRITING PROCESS
   7 guide to computers a tool in the writing process/ 1984  bks
OPTIONS: ----------------- More - to see next page ---------------------------
                                                                        Help
 GUide           display # - see #th item                               Quit
 Help COMMANDS   REDo - edit search          STOre # - save for email   COMment
COMMAND?
```

Screen 10



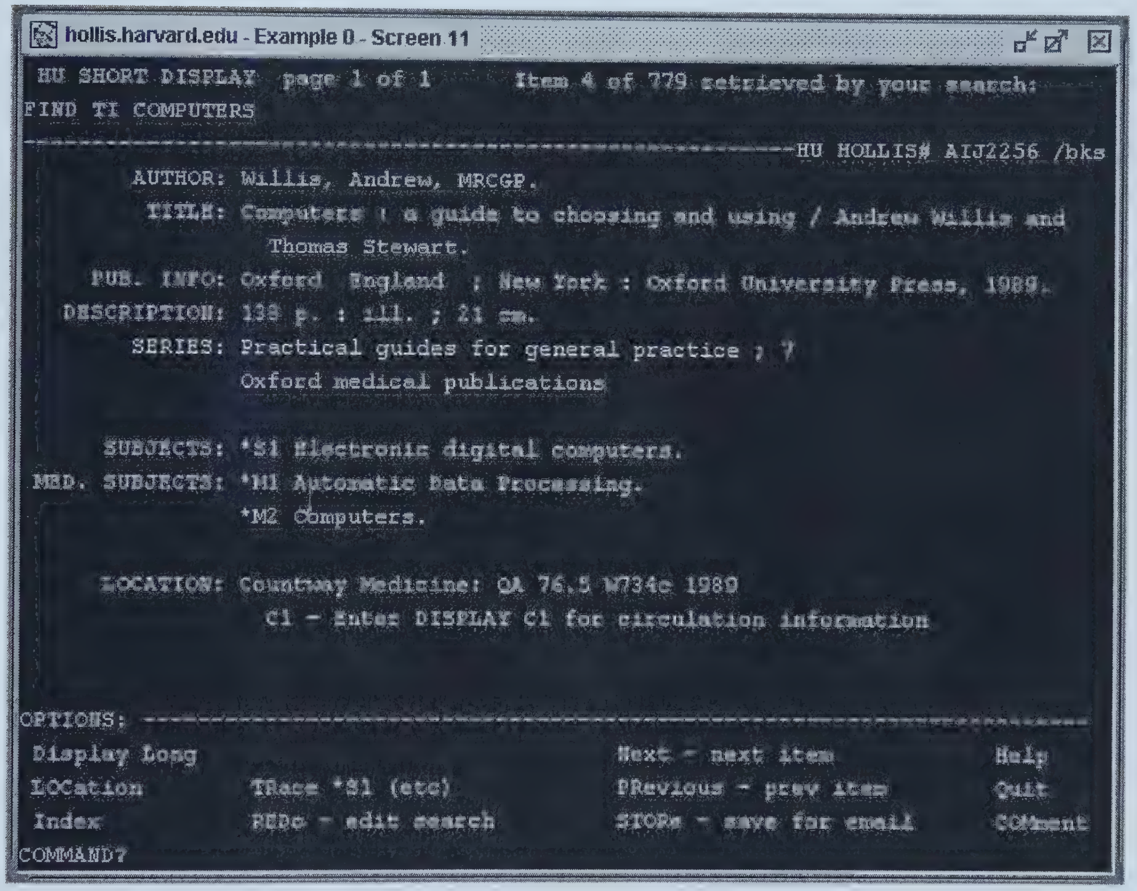
```
 HU SHORT DISPLAY  page 1 of 1      Item 3 of 779 retrieved by your search:
FIND TI COMPUTERS
-----------------------------------------------------------HU HOLLIS# API1001 /bks
        AUTHOR: Fielker, David, 1932-
         TITLE: Computers / David S. Fielker.
      PUB. INFO: Cambridge  Eng.  : University Press, 1967.
    DESCRIPTION: 32 p. : ill. ; 22 cm.
        SERIES: Topics from mathematics

      SUBJECTS: *S1 Computers.
                *S2 Electronic data processing.

      LOCATION: Gutman Education: QA76.F5
                  C1 - Enter DISPLAY C1 for circulation information




OPTIONS: -------------------------------------------------------------------
 Display Long                           Next - next item             Help
 LOCation        TRace *S1 (etc)        PRevious - prev item         Quit
 Index           REDo - edit search     STOPs - save for email       COMment
COMMAND?
```

Screen 11



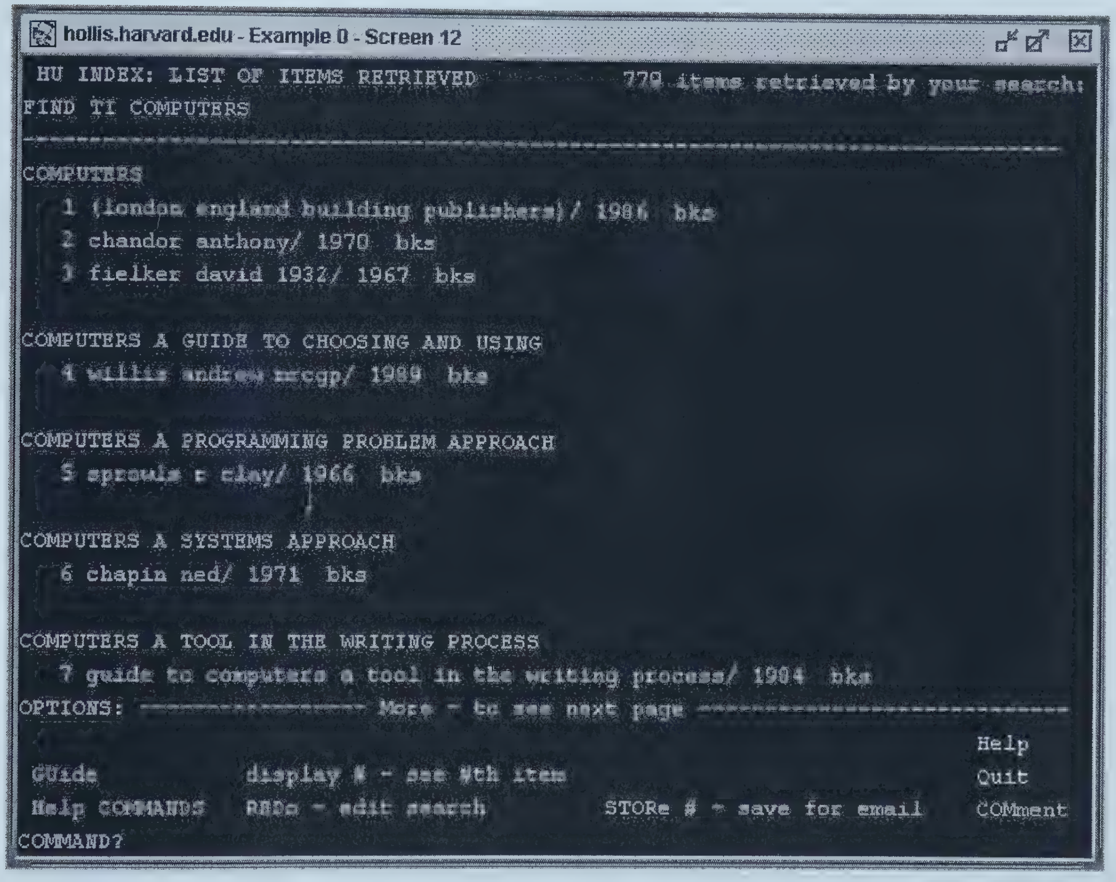
```
 HU INDEX: LIST OF ITEMS RETRIEVED            779 items retrieved by your search:
FIND TI COMPUTERS
--------------------------------------------------------------------------------
COMPUTERS
   1 (london england building publishers)/ 1986  bks
   2 chandor anthony/ 1970  bks
   3 fielker david 1932/ 1967  bks

COMPUTERS A GUIDE TO CHOOSING AND USING
   4 willis andrew mrcgp/ 1989  bks

COMPUTERS A PROGRAMMING PROBLEM APPROACH
   5 sprowls r clay/ 1966  bks

COMPUTERS A SYSTEMS APPROACH
   6 chapin ned/ 1971  bks

COMPUTERS A TOOL IN THE WRITING PROCESS
   7 guide to computers a tool in the writing process/ 1984  bks
OPTIONS: ----------------- More - to see next page -------------------------
                                                                       Help
 GUide           display # - see #th item                              Quit
 Help COMMANDS   REDo - edit search        STORe # - save for email    COMment
COMMAND?
```

Screen 12



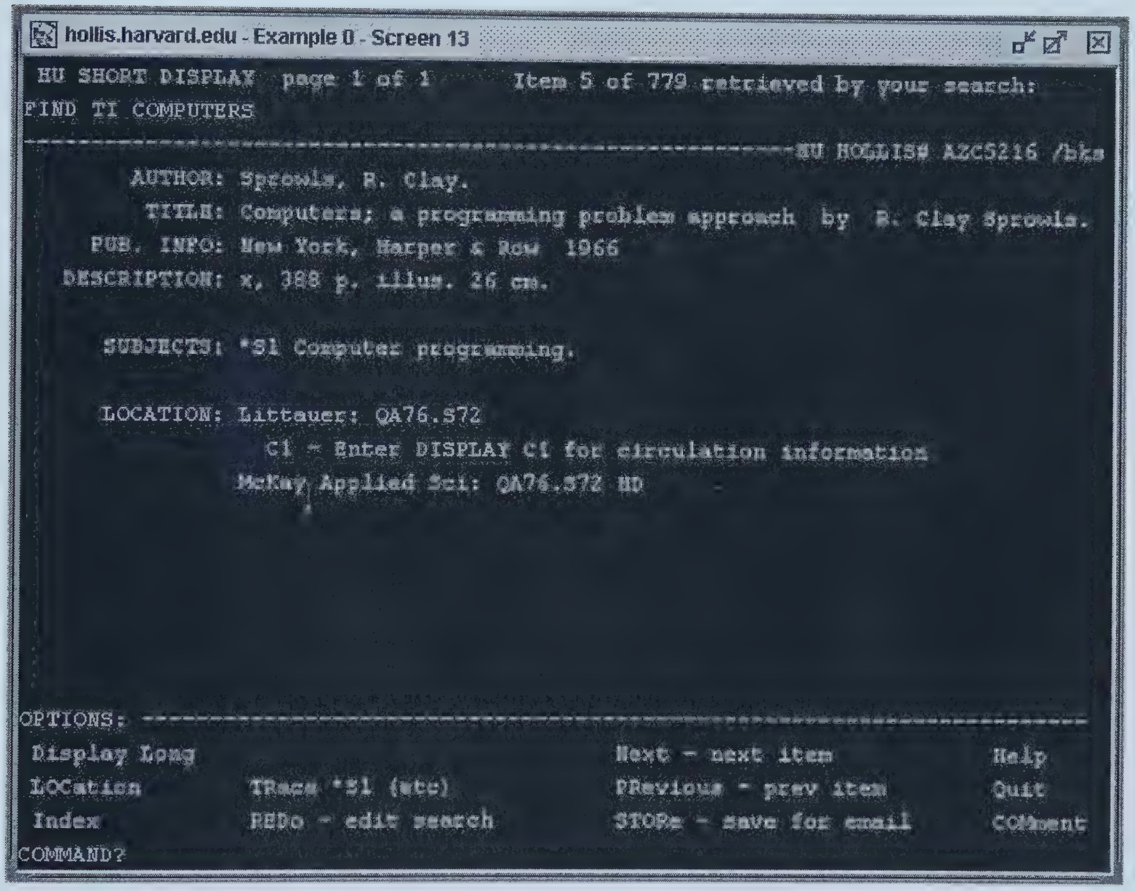
```
 HU SHORT DISPLAY  page 1 of 1      Item 4 of 779 retrieved by your search:
FIND TI COMPUTERS
-------------------------------------------------------HU HOLLIS# AIJ2256 /bks
        AUTHOR: Willis, Andrew, MRCGP.
         TITLE: Computers : a guide to choosing and using / Andrew Willis and
                  Thomas Stewart.
     PUB. INFO: Oxford  England  ; New York : Oxford University Press, 1989.
   DESCRIPTION: 138 p. : ill. ; 21 cm.
        SERIES: Practical guides for general practice ; 7
                Oxford medical publications

      SUBJECTS: *S1 Electronic digital computers.
 MED. SUBJECTS: *M1 Automatic Data Processing.
                *M2 Computers.

      LOCATION: Countway Medicine: QA 76.5 W734c 1989
                  C1 - Enter DISPLAY C1 for circulation information


OPTIONS: ---------------------------------------------------------------------
 Display Long                               Next - next item            Help
 LOCation         TRace *S1 (etc)           PRevious - prev item        Quit
 Index            REDo - edit search        STORe - save for email      COMment
COMMAND?
```

Screen 13



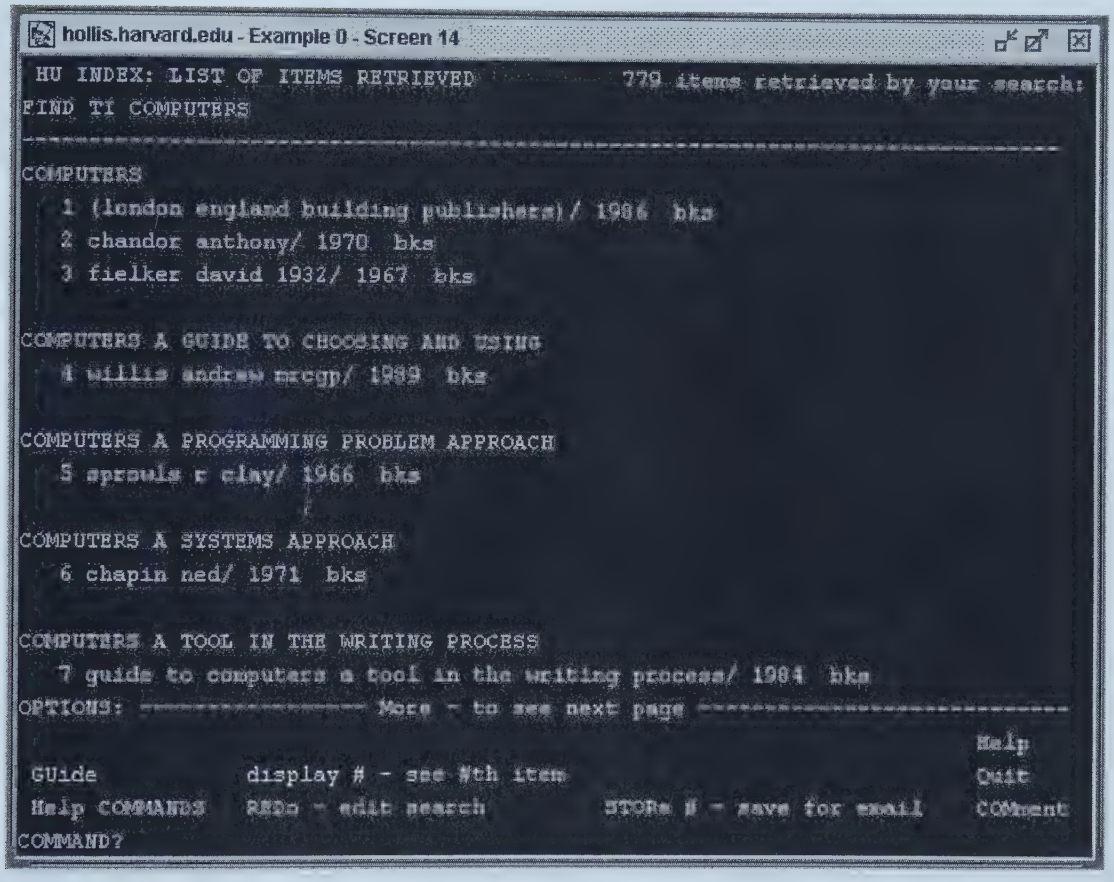
```
 HU INDEX: LIST OF ITEMS RETRIEVED           779 items retrieved by your search:
FIND TI COMPUTERS
--------------------------------------------------------------------------------
COMPUTERS
   1 (london england building publishers)/ 1986  bks
   2 chandor anthony/ 1970  bks
   3 fielker david 1932/ 1967  bks

COMPUTERS A GUIDE TO CHOOSING AND USING
   4 willis andrew mrcgp/ 1989  bks

COMPUTERS A PROGRAMMING PROBLEM APPROACH
   5 sprowls r clay/ 1966  bks

COMPUTERS A SYSTEMS APPROACH
   6 chapin ned/ 1971  bks

COMPUTERS A TOOL IN THE WRITING PROCESS
   7 guide to computers a tool in the writing process/ 1984  bks
OPTIONS: ----------------- More - to see next page ----------------------------
                                                                    Help
 GUide           display # - see #th item                           Quit
 Help COMMANDS   REDo - edit search        STORe # - save for email  COMment
COMMAND?
```

Screen 14



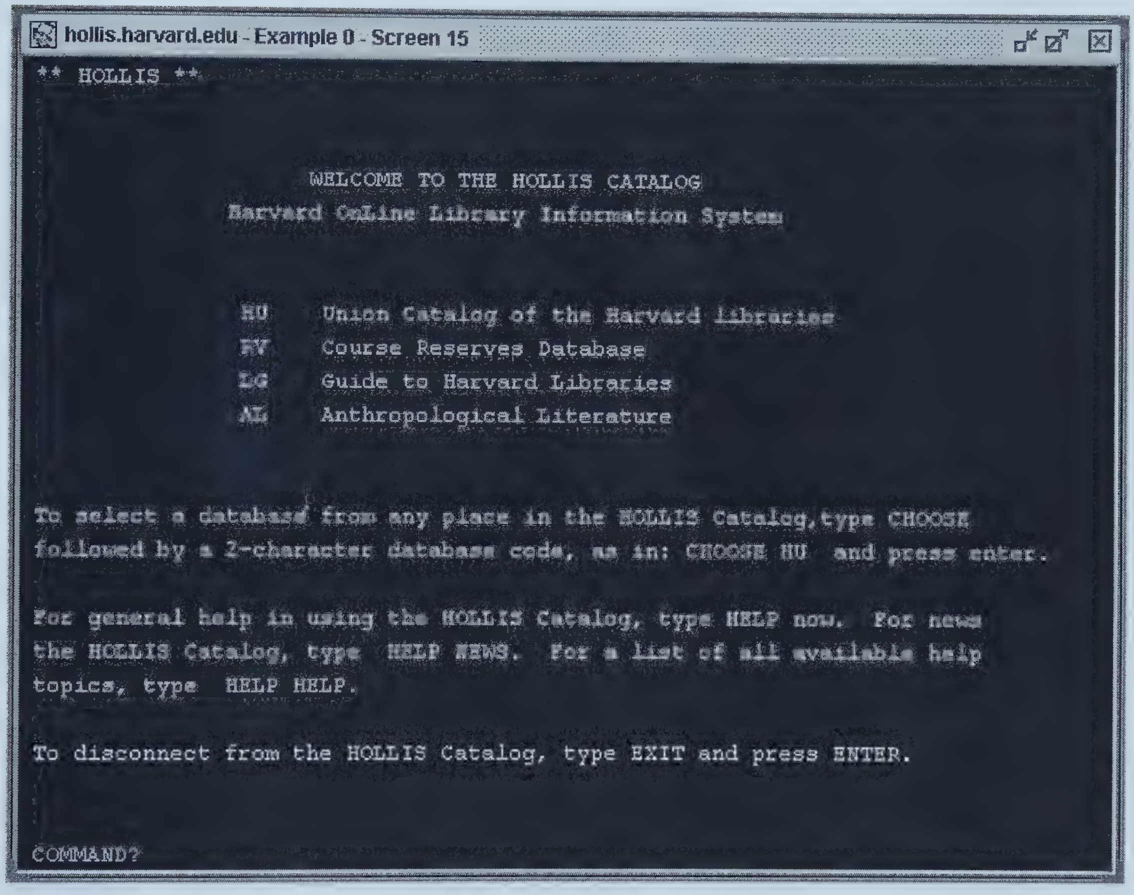
```
 HU SHORT DISPLAY  page 1 of 1      Item 5 of 779 retrieved by your search:
FIND TI COMPUTERS
-------------------------------------------------------------HU HOLLIS# AZC5216 /bks
        AUTHOR: Sprowls, R. Clay.
         TITLE: Computers; a programming problem approach  by  R. Clay Sprowls.
      PUB. INFO: New York, Harper & Row  1966
   DESCRIPTION: x, 388 p. illus. 26 cm.

      SUBJECTS: *S1 Computer programming.

      LOCATION: Littauer: QA76.S72
                  C1 - Enter DISPLAY C1 for circulation information
                McKay Applied Sci: QA76.S72 HD





OPTIONS: ---------------------------------------------------------------------------
 Display Long                               Next - next item            Help
 LOCation        TRace *S1 (etc)            PRevious - prev item        Quit
 Index           REDo - edit search         STORe - save for email      COMment
COMMAND?
```

Screen 15



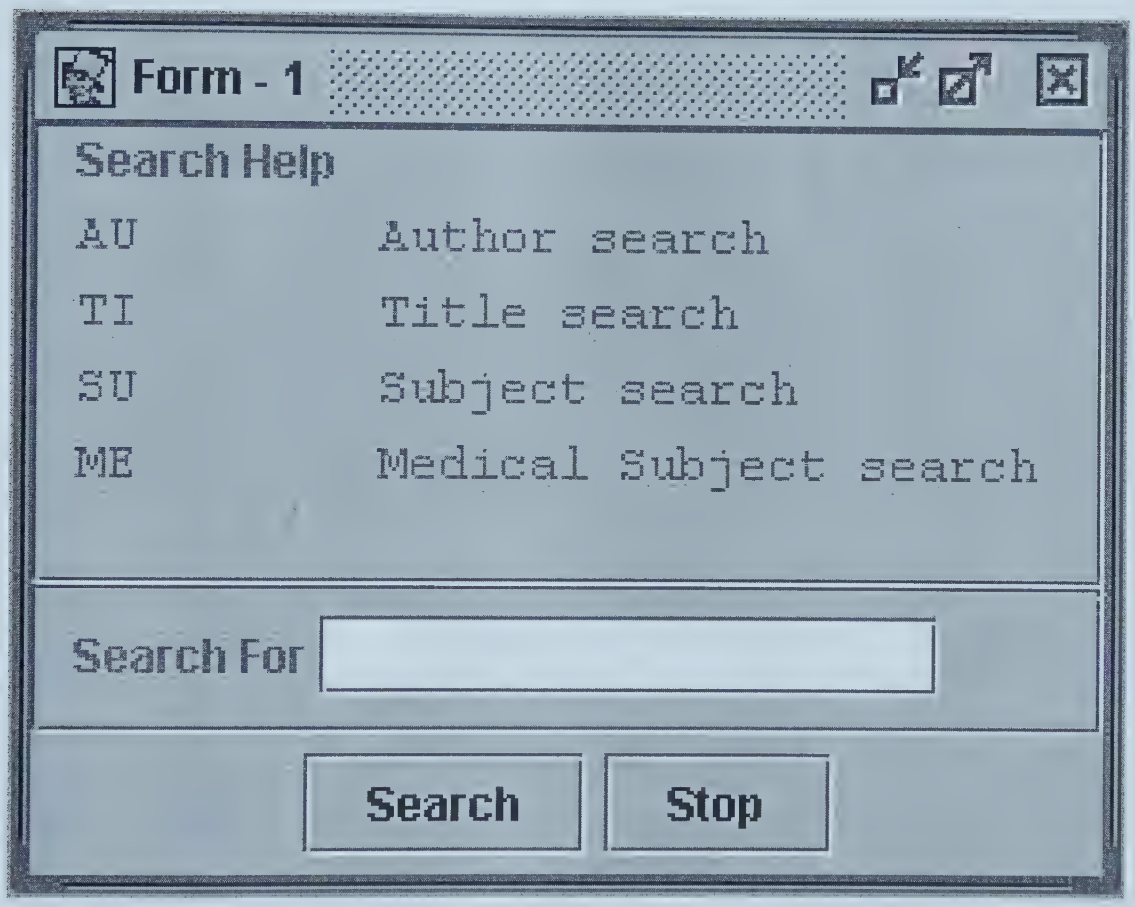
```
 HU INDEX: LIST OF ITEMS RETRIEVED           779 items retrieved by your search:
FIND TI COMPUTERS
------------------------------------------------------------------------------
COMPUTERS
   1 (london england building publishers)/ 1986  bks
   2 chandor anthony/ 1970  bks
   3 fielker david 1932/ 1967  bks

COMPUTERS A GUIDE TO CHOOSING AND USING
   4 willis andrew mrcgp/ 1989  bks

COMPUTERS A PROGRAMMING PROBLEM APPROACH
   5 sprowls r clay/ 1966  bks

COMPUTERS A SYSTEMS APPROACH
   6 chapin ned/ 1971  bks

COMPUTERS A TOOL IN THE WRITING PROCESS
   7 guide to computers a tool in the writing process/ 1984  bks
OPTIONS: ---------------- More - to see next page -------------------------
                                                                      Help
 GUide            display # - see #th item                            Quit
 Help COMMANDS    REDo - edit search         STORe # - save for email  COMment
COMMAND?
```

Screen 16



```
** HOLLIS **


                WELCOME TO THE HOLLIS CATALOG
            Harvard OnLine Library Information System


             HU    Union Catalog of the Harvard Libraries
             RV    Course Reserves Database
             LG    Guide to Harvard Libraries
             AL    Anthropological Literature


To select a database from any place in the HOLLIS Catalog,type CHOOSE
followed by a 2-character database code, as in: CHOOSE HU  and press enter.

For general help in using the HOLLIS Catalog, type HELP now.  For news
the HOLLIS Catalog, type  HELP NEWS.  For a list of all available help
topics, type  HELP HELP.

To disconnect from the HOLLIS Catalog, type EXIT and press ENTER.


COMMAND?
```

Abstract GUI Form 1



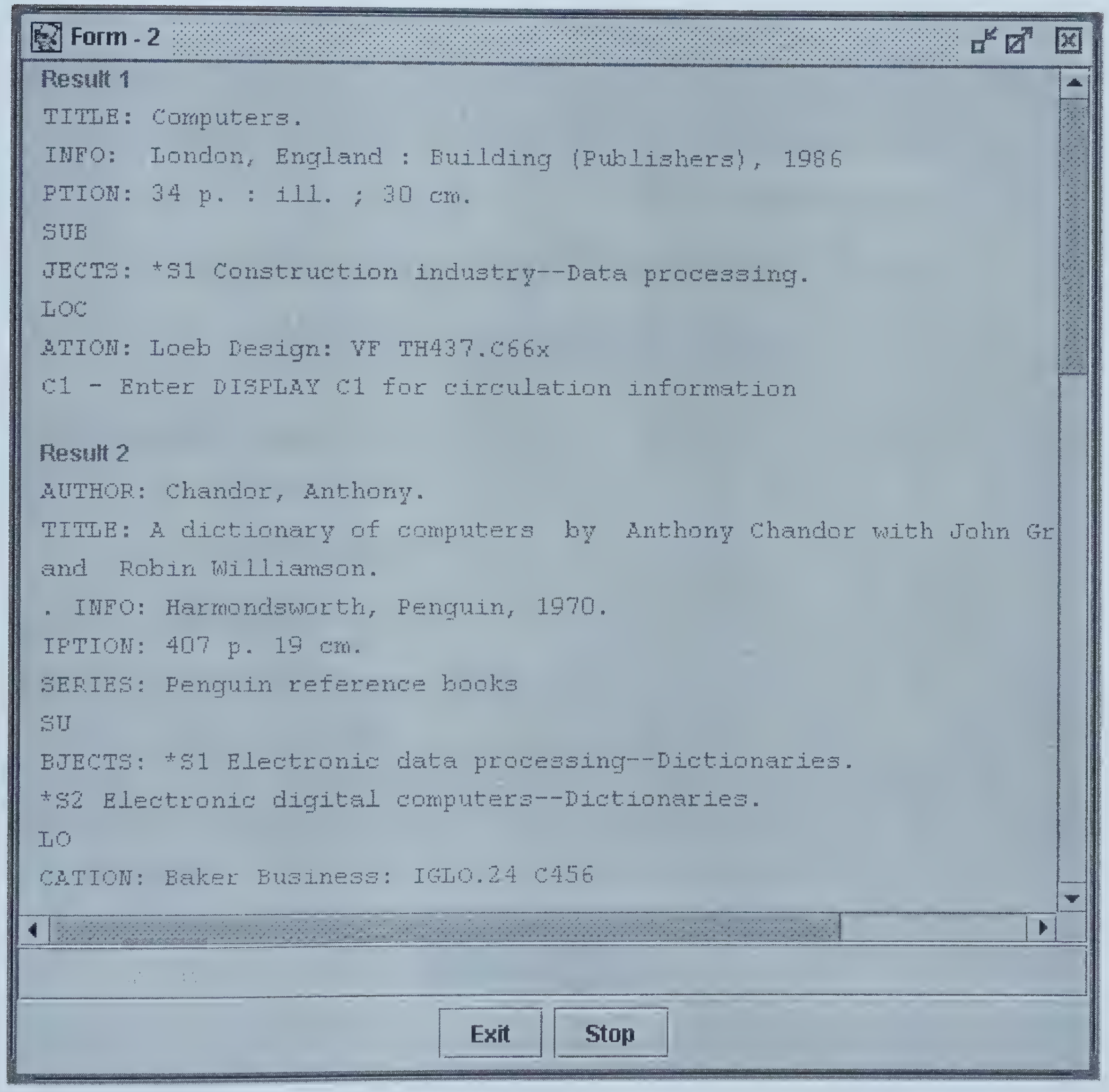
Form - 1

Search Help

AU	Author search
TI	Title search
SU	Subject search
ME	Medical Subject search

Search For

Abstract GUI Form 2



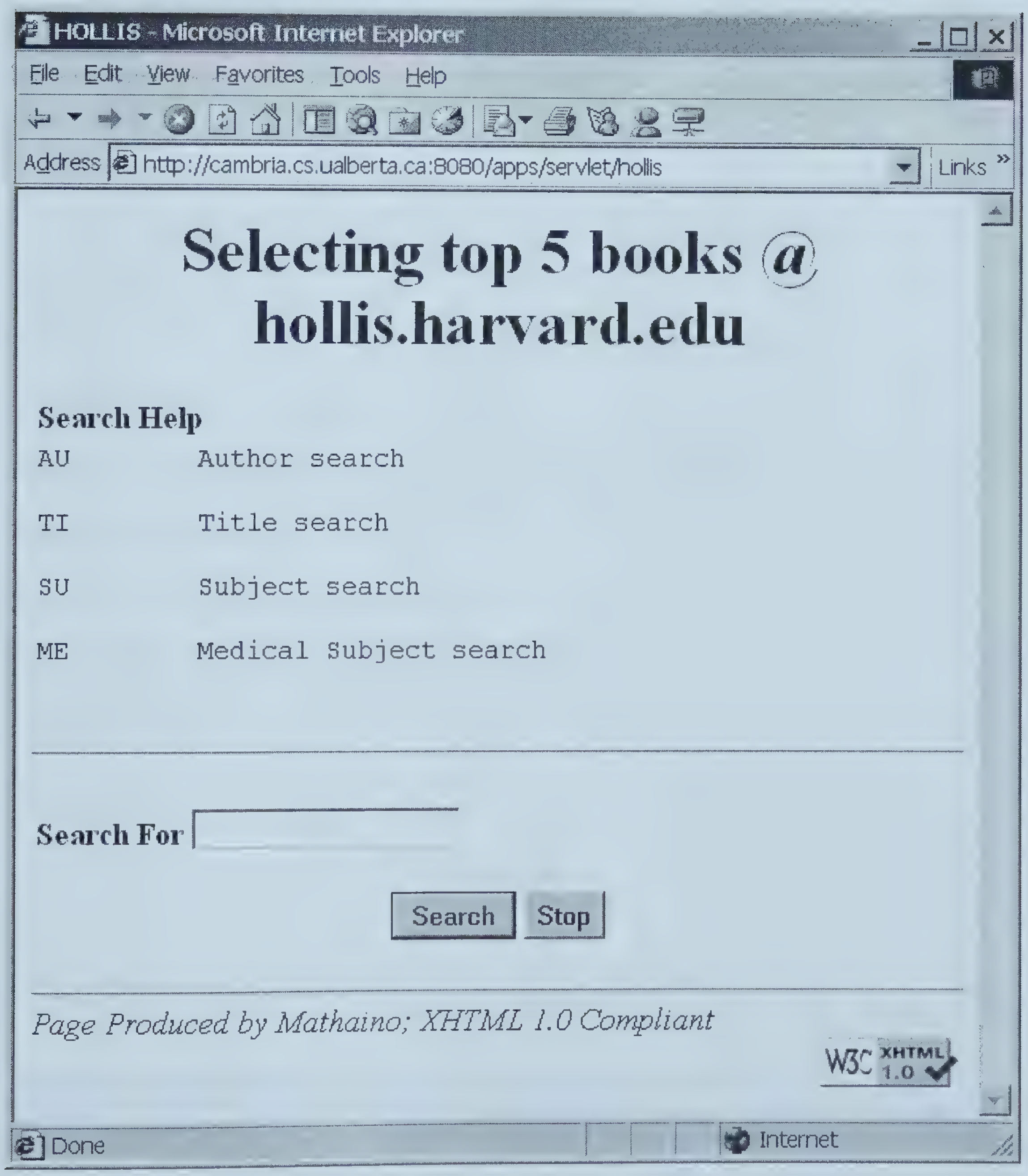
Result 1

```
TITLE: Computers.
INFO:  London, England : Building (Publishers), 1986
PTION: 34 p. : ill. ; 30 cm.
SUB
JECTS: *S1 Construction industry--Data processing.
LOC
ATION: Loeb Design: VF TH437.C66x
C1 - Enter DISPLAY C1 for circulation information
```

Result 2

```
AUTHOR: Chandor, Anthony.
TITLE: A dictionary of computers  by  Anthony Chandor with John Gr
and  Robin Williamson.
. INFO: Harmondsworth, Penguin, 1970.
IPTION: 407 p. 19 cm.
SERIES: Penguin reference books
SU
BJECTS: *S1 Electronic data processing--Dictionaries.
*S2 Electronic digital computers--Dictionaries.
LO
CATION: Baker Business: IGLO.24 C456
```

XHTML Form 1



http://cambria.cs.ualberta.ca:8080/apps/servlet/hollis

Selecting top 5 books @ hollis.harvard.edu

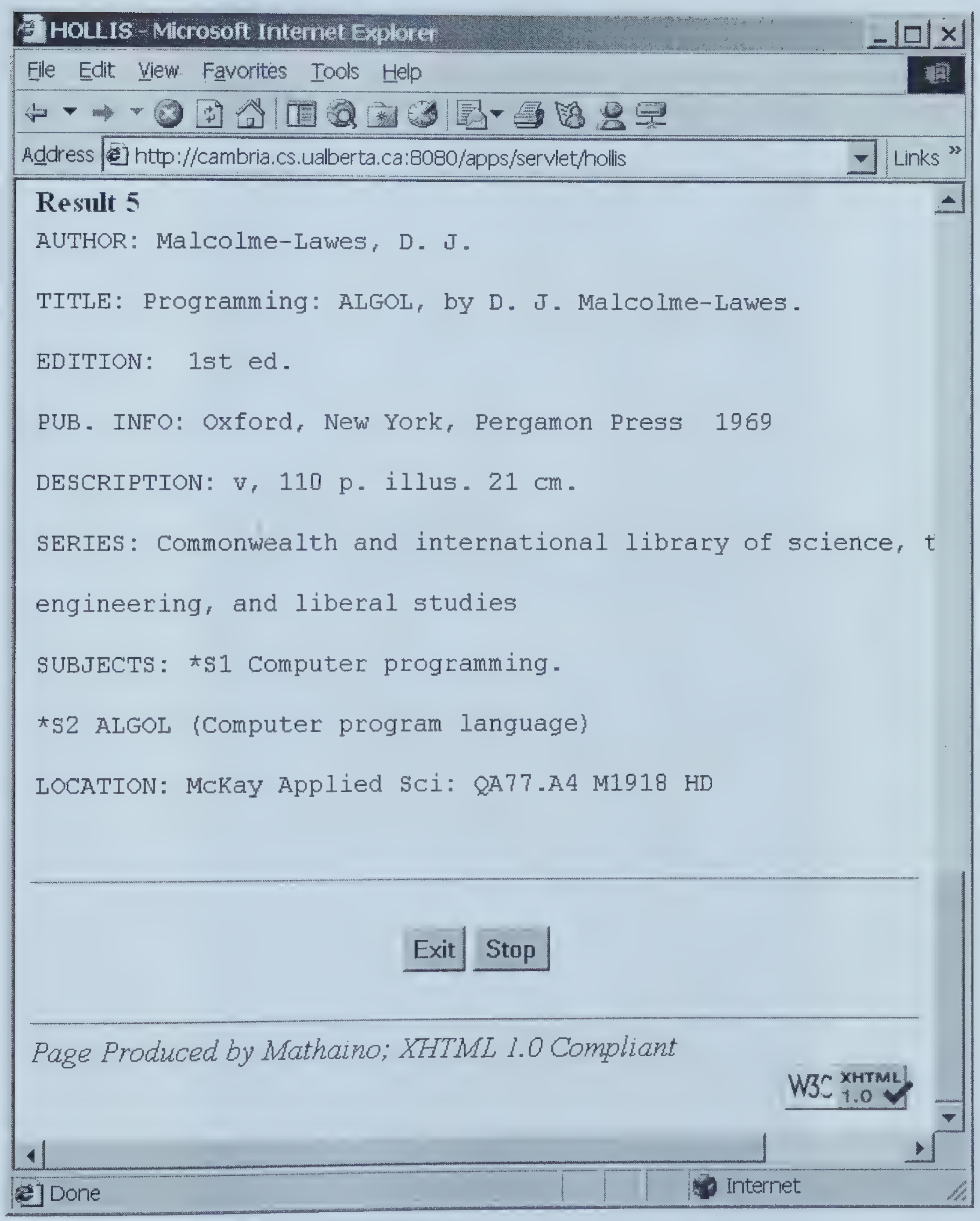
Search Help

AU	Author search
TI	Title search
SU	Subject search
ME	Medical Subject search

Search For

Page Produced by Mathaino; XHTML 1.0 Compliant

XHTML Form 2



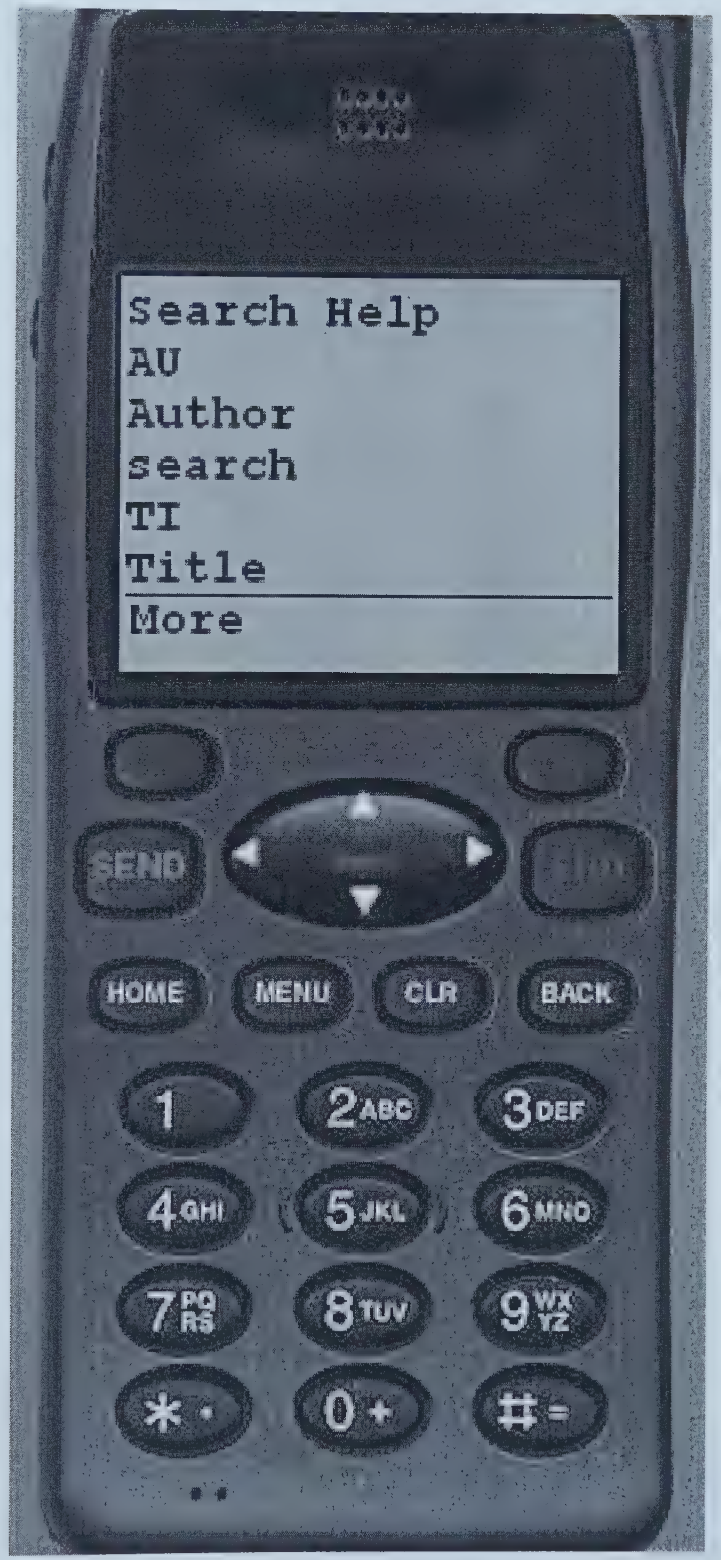
HOLLIS - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://cambria.cs.ualberta.ca:8080/apps/servlet/hollis
Links »

Result 5

```
AUTHOR: Malcolme-Lawes, D. J.

TITLE: Programming: ALGOL, by D. J. Malcolme-Lawes.

EDITION:  1st ed.

PUB. INFO: Oxford, New York, Pergamon Press  1969

DESCRIPTION: v, 110 p. illus. 21 cm.

SERIES: Commonwealth and international library of science, t

engineering, and liberal studies

SUBJECTS: *S1 Computer programming.

*S2 ALGOL (Computer program language)

LOCATION: McKay Applied Sci: QA77.A4 M1918 HD
```

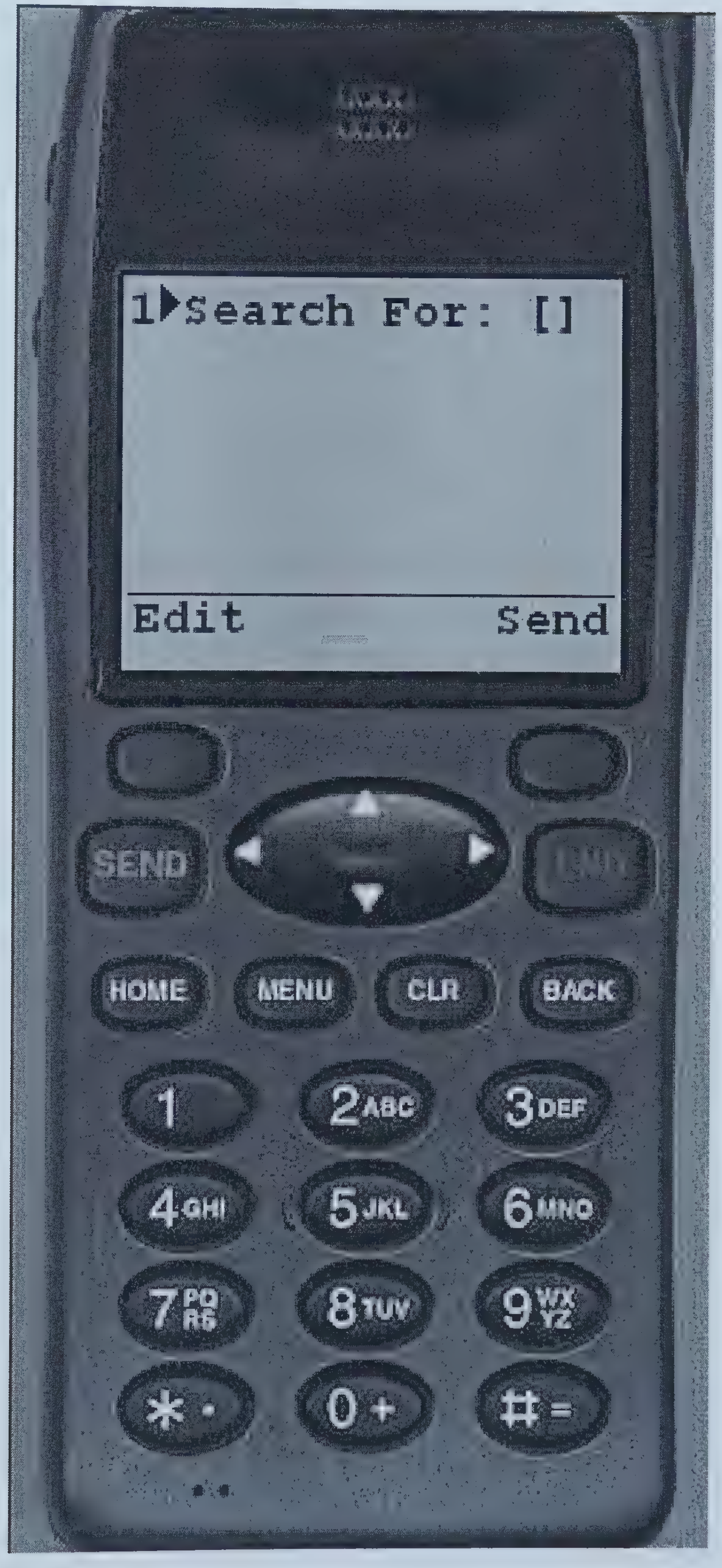
Page Produced by Mathaino; XHTML 1.0 Compliant

WML Form 1 (Abstract Form 1 Part 1)



Search Help
AU
Author
search
TI
Title
More

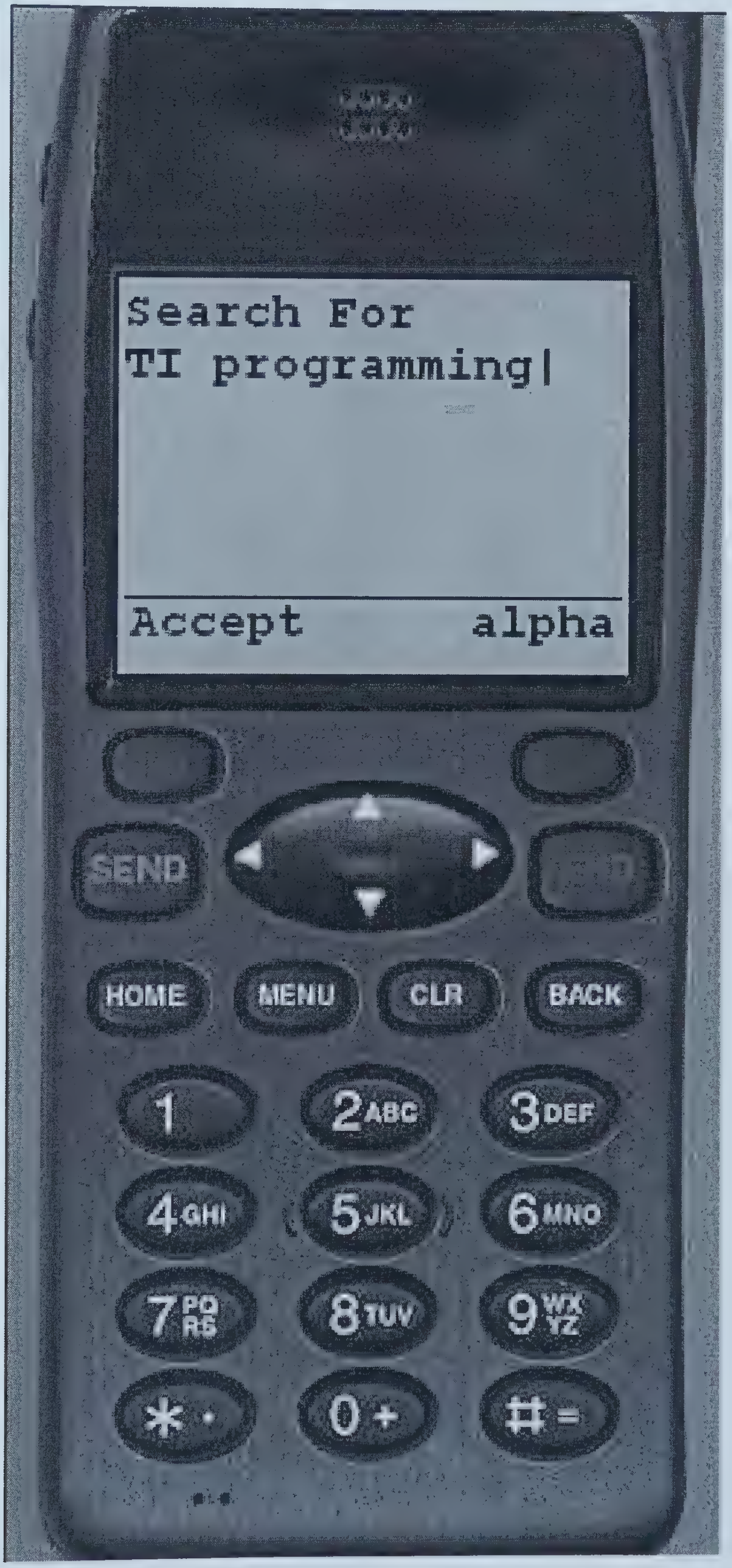
WML Form 2 (Abstract Form 1 Part 2)



1▶Search For: []

Edit Send

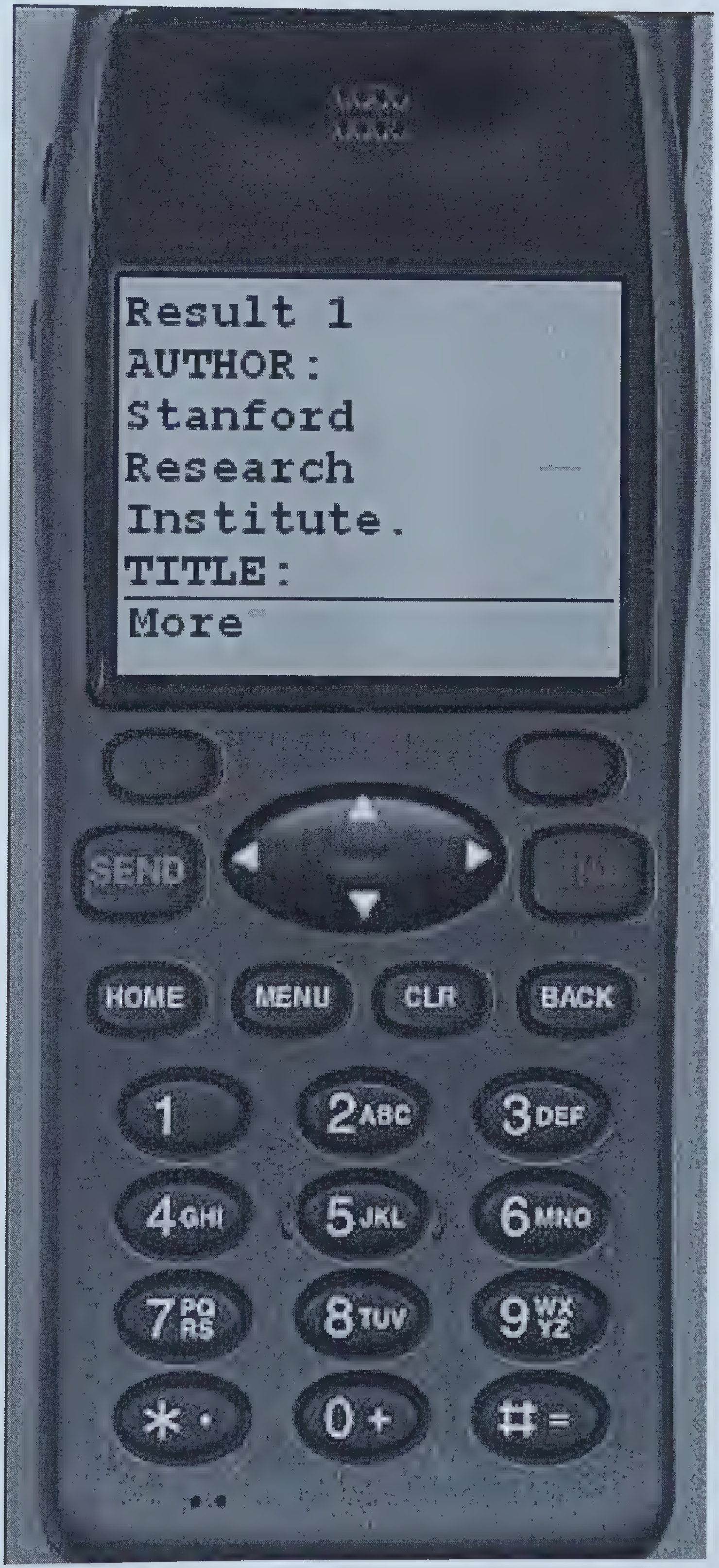
WML Form 3 (Abstract Form 1 Part 3, WML Input Widget)



Search For
TI programming|

Accept alpha

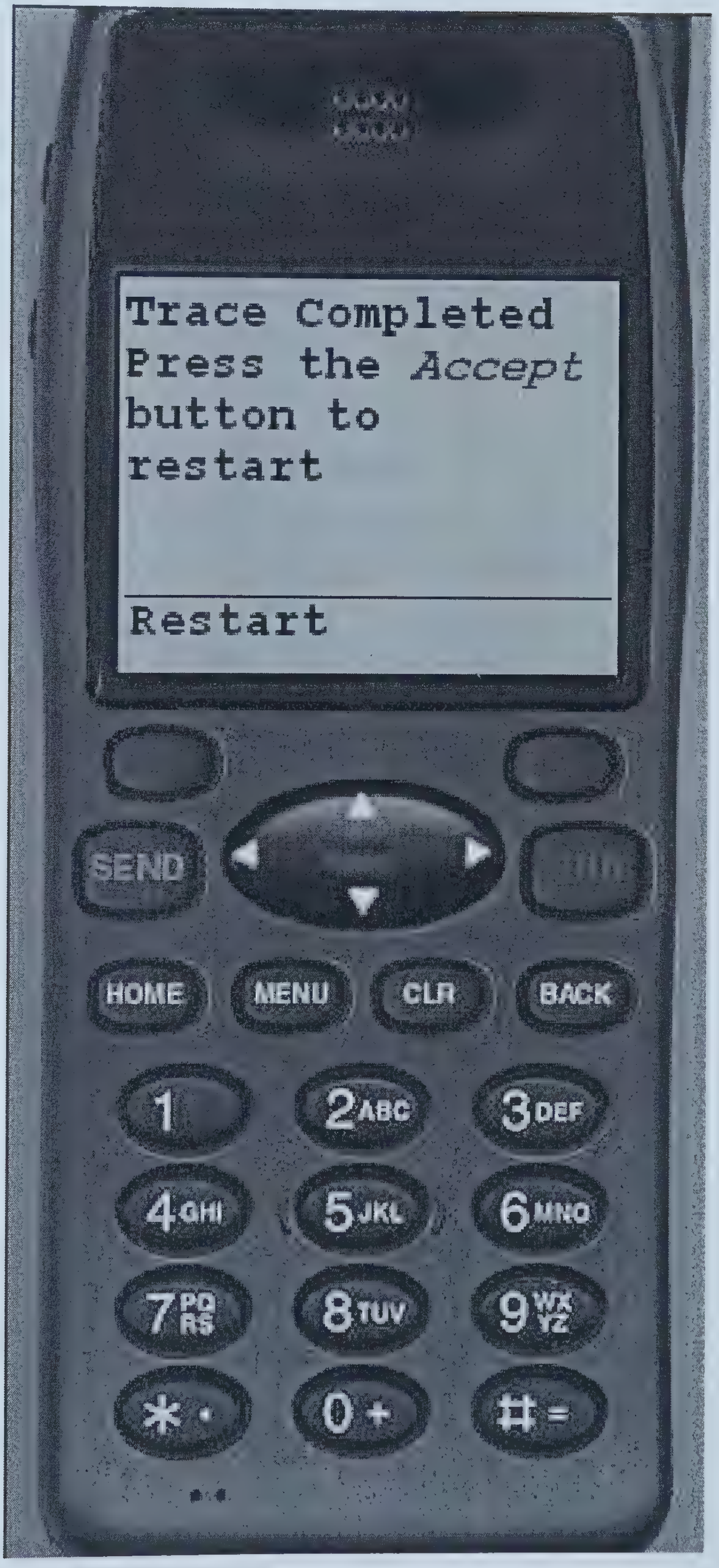
WML Form 4 (Abstract Form 2 Part 1)



Result 1
AUTHOR:
Stanford
Research
Institute.
TITLE:

More

WML Form 5 (WML Exit Form)



Trace Completed
Press the *Accept*
button to
restart

Restart

Appendix B

Coding Conventions

1. Why Have Coding Conventions?

Coding conventions are important to programmers for a number of reasons:

- 80% of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
- If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

2. Organization of a JAVA class

The code sample in Figure A2-1 lists an empty Java class.

```
class ClassName  // cn
{
////////////////////////////////////////
// constructors
////////////////////////////////////////
      public ClassName() {}


////////////////////////////////////////
// operations ( instance methods )
////////////////////////////////////////
      public void doSomething() {}


////////////////////////////////////////
// implementation ( instance methods )
////////////////////////////////////////
      private void doPrivateThing() {}


////////////////////////////////////////
// implementation ( class methods )
////////////////////////////////////////
      private static void doStaticThing() {}


////////////////////////////////////////
// implementation ( instance variables )
////////////////////////////////////////
      private int m_nMyPrivateNumber;


////////////////////////////////////////
// implementation ( class constants )
////////////////////////////////////////
      private static final String S_MY_STRING;
}
```

Figure A2-1: A Sample Java Class

All Java classes should be organized in accordance with this skeleton. This clearly separates each class into easily understandable logical blocks. Such a separation is also in accordance with OO principles as it clearly lays out the public interface of the class and separates it from the private details. The following section defines what exactly constitutes each logical block in a class.

- **Constructors:** These are the normal constructors to be used while creating objects of this class.

- **Operations:** These are the various operations that can be performed on the objects of this class. The set of functions in this category constitutes the public interface of the class.

- **Implementation:** Functions and variables declared under an implementation section are private (or protected) to a class, and are not exported externally.

- **Class Methods:** Static member functions of a class are also known as class methods.

- **Instance Methods:** Non-static member functions of a class are also known as instance methods.

- **Class Variables:** Static member variables of a class are also known as class variables.

- **Class Constants:** Static and final member variables of a class constitute the set of class constants.

3. Naming Guidelines

Table A2-1 lists the naming guidelines for Java. In most cases it is simply a copy of the corresponding guidelines issued by Sun with minor modifications.

Identifier Type	Rules for Naming	Example
Packages	The prefix of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981. Subsequent components of the package name vary according to an organization's own internal naming conventions. Such conventions might specify that certain directory name components be division, department, project, machine, or login names.	`com.sun.eng` `ca.ualberta.cs.cellest`

Identifier Type	Rules for Naming	Example

Classes	Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words-avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).
	Also clearly declare the mnemonic to be used for naming the variables of this class using a comment immediately following its name. For example, the mnemonic for the class listed in Example 1 is `cn`.
		`class Raster //raster {}`
		class ImageSprite //is {}

Interfaces	Interface names should be capitalized like class names.	`interface Storing;`

Identifier Type	Rules for Naming	Example

Methods	Method names should be verbs, in mixed case with the first letter lowercase, and the first letter of each internal word capitalized.
	As usually a class doesn't export any internal variables, there should be a pair of get/set methods for each attribute variable of the class and the first word of their name must be either get or set respectively.
	Any method that returns the value of some boolean flag must have the word is as the first word of its name.
	Any method that returns size of some quantity must have the word length as the first word of its name.
		doSomething();
		getColor();
		setColor();
		isThisTrue();
		length();

Identifier Type	Rules for Naming	Example

Variables	All variable names are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore _ or dollar sign $ characters, even though both are allowed.
	Variable names should be short yet meaningful. The choice of a variable name should be mnemonic- that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.
	All instance variables should start with the sequence m_. This indicates that they are member variables of the class and clearly distinguishes them from other local variables.
	Each variable name should start with a lowercase mnemonic that clearly identifies its type. Table 1 lists the starting mnemonic for each Java type. When needed multiple mnemonics should be combined while naming the variables.
		int i;
		char c;
		float dWidth;
		int m_nAge;
		String m_sName;
		Vector m_vector_sNames;
		MyClass mcMyClassObj;

Identifier Type	Rules for Naming	Example

Constants	The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("_"). (ANSI constants should be avoided, for ease of debugging.)
	Also the first word of the name should be an appropriate mnemonic for its type.
		`static final int N_WIDTH = 4;`
		`static final String S_NAME = "CELLEST";`

Table A2-1: Java Naming Guidelines

Variable Type	Mnemonic	Explanation
Int	n	n stands for number
Byte	b	b stands for byte
Char	c	c stands for char
boolean	f	f stands for flag
array	p	p stands for pointer to
word	w	w stands for word
long	l	l stands for long
unsigned int	u	u stands for unsigned
double/float	d	d stands for double
String	s	s stands for string
user defined classes	Appropriate mnemonic indicated in their declaration	
Other Java API classes	The name of the class in lowercase. e.g. `Jframe jframeMain;`	

Table A2-2: Mnemonics for Java Data Types

The mnemonics for various Java data types are listed in table A2-2.

Examples of some variable names

`m_nCounter`	A member variable of type integer being used as a counter
`nCounter`	A non-member integer variable being used as a counter
`sName`	A string variable containing a name

`vector_sName`	A vector of strings for holding names
`m_vector_cn`	A member vector of objects of type ClassName (see Example 1)
`psName`	A simple array of strings containing names
`m_psName`	A member array of strings containing names

4. Indentation

Four spaces should be used as the unit of indentation. Avoid lines longer than 80 characters, since they're not handled well by many terminals and tools. You can indent your code in two ways:

1. Old Kernighan and Ritchie (K&R) Style

```
public class MyClass {  // mc
      ...
      private void doSomething {
            ...
            if ( fIsTrue ) {
                  ...
                  ...
            }
            ...
      }
      ...
      ...
}
```

2. Modern OO Style

```
public class MyClass  // mc
{
      ...
      private void doSomething
      {
            ...
            if ( fIsTrue )
            {
                  ...
                  ...
            }
            ...
      }
      ...
      ...
}
```

The K&R style was first introduced by Kernighan and Ritchie in their famous book *The C Programming Language*. Many programmers who have migrated to Java/C++ from C use this style even today. I, personally, prefer the modern OO style, but this is a matter of choice. Modern IDEs usually allow the user to chose one style or the other. Both of them are equally good provided they are followed consistently throughout the entire project.

5. Javadoc Comments

The JAVA programming language includes the javadoc tool to aid programmers in documenting their programs. Any valid HTML tags can be incorporated within all javadoc comments. The following guidelines should be adopted for proper documentation of Java programs.

Documenting Classes

Classes should be documented using the following guidelines

- A short paragraph or two describing the purpose and role of the class should precede each class declaration.

- It is also recommended that the author, version and date of creation of the class be recorded as a part of this documentation. Javadoc tags `@author` and `@version` should be used for this purpose. Recording the proper role of the class in the current project also yields better program designs.

Example

```
import java.lang.*;

/**
 * This is an example class comment. It should be document
 * the purpose, role and any other important features of
 * this class.
 * <p>
 * Created: Thu Jun 22 15:41:14 2000
 * <p>
 * @author PQR XYZ
 * @version 1.0
 *
 */

public class ExampleClass  // exc
{ }
```

Notice that the class comment needs to be placed after the `import` statements, otherwise the Javadoc tool will not recognize it properly.

Documenting Class Methods

Class methods should be documented using the following guidelines

- Each method declaration should be preceded by a javadoc comment approximately a couple of lines to a paragraph in length describing exactly what that method accomplishes.

- If the method takes any input parameters, then there should be a javadoc comment indicated by the `@param` javadoc tag describing them.

- If the method returns anything then again, there should be an appropriate javadoc comment describing exactly what is being returned. The `@return` javadoc tag should be used for this purpose.

- If the method throws any exception, then that fact should be highlighted by using the `@throws` javadoc tag.

Example

```
/**
 * This function adds the given node to the internal vector.
 * <p>
 * @param fIsWildcard The wildcard flag for this node
 * @param sFixedWord The value for the fixed word, if required
 * @param sWildcardValue The value for the wildcard, if required
 * <p>
 * @return <code>true</code> on success and <code>false</code>
 *         on failure
 *
 */
public boolean addTemplateNode( final boolean fIsWildcard,
      final String sFixedWord,
      final String sWildcardValue )
{
      ...
      ...
}
```

Documenting Member Instance Variables

By default no comments are necessary for class constants (i.e. final member class variables) or local variables. Only member instance variables need an explanation. The easiest way to accomplish this is by using the `@serial` javadoc tag followed by a single line of comment describing the role of that instance variable in the current class. For more information about the `@serial` tag and its variations please refer to [JAVA99].

Example

```
/** @serial The vector containing the names of people */
private String m_vector_sNames;
```

6. Passing Parameters to Functions and Some JAVA Patterns to Avoid

In JAVA all objects are passed by reference to functions. This is same as passing objects by reference in C++. It is also very similar to passing objects using pointers in C as the value of the object being passed can be changed by the function.

So it is necessary to declare the parameters being passed as `final` so that they are not accidentally modified during the execution of the function. (This apparently does not work, as the Java compiler is unable to detect that member functions can change the state of an object, but we can always hope that newer versions of the *javac* compiler will fix this bug)

Example

```
public void doSomething( final Vector vector_sName )
{
      ...
      ...
}
```

Also, although undocumented, the assignment operator in JAVA too does a similar thing, i.e. all assignments in JAVA are by reference. Even if the `new` keyword is used this rule holds as the variable is assigned a reference of the object just created. Thus, the following JAVA and C++ statements are equivalent:

```
C++:  MyCoolClass& mccVariable = mccTheOriginalObject;

JAVA: MyCoolClass mccVariable = mccTheOriginalObject;
```

Note the following C++ statement is not equivalent to above as here the copy constructor of the given class would be invoked and the variable `mccVariable` will hold an exact replica of the object `mccTheOriginalObject` but will not actually be a reference to it.

```
C++: MyCoolClass mccVariable = mccTheOriginalObject;
```

Now the reason it was important to point out this difference is that in case of JAVA if any changes are made to `mccVariable` then as it simply refers to `mccTheOriginalObject`, the state of that object would be modified too.

So the following code pattern should be avoided in JAVA:

```
public void doSomething( final SomeObject soOriginal )
{
      m_so = soOriginal;
      return;
}
```

This is because if this were done, then the external object's (`soOriginal`) state would be modified every time the internal variable (`m_so`) is modified. Even worse, the internal variable's (`m_so`) state would be modified every time the original object was modified, violating the fact that its state must not be changed by external entities (so much for encapsulation etc.). The fact that using the word `final` does not do anything to help this situation makes it even more dangerous.

The proper way to handle this situation is by using the following code pattern instead:

```
public void doSomething( final SomeObject soOriginal )
{
      m_so = new SomeObject( soOriginal );
      return;
}
```

Now the internal object (m_so) is independent of the external object (soOriginal). Another implication of this is that most JAVA classes will now need copy constructors as the statement "new SomeObject(soOriginal); " will not work without an appropriate copy constructor for the class SomeObject.

Again for the same reason avoid the following code pattern:

```
public SomeObject getSomething()
{
      return m_so;
}
```

Instead use:

```
public SomeObject getSomething()
{
      return new SomeObject( m_so );
}
```

7. Appropriate Use of `import` Statements

Just like the `include` statements in C/C++ the `import` statements in Java should directly include all the files containing the classes that are being used in the current piece of code. Indirect `import` statements should be avoided as they decrease the readability of code.

Thus, avoid the following:

`File: ClassC.java`

```
import ClassB;

public class ClassC
{
      ...
      private ClassA m_classaObject;
      private ClassB m_classbObject;
}
```

This is because if `ClassC` uses an object of type `ClassA` then it should directly import it and not rely on the fact that the compiler will find that file anyway because of the `classpath` or the `PATH` variable. (Note here I am talking about classes but the same applies to packages too)

So instead the file for `ClassC` should be coded as:

```
File: ClassC.java

import ClassA;
import ClassB;

public class ClassC
{
      ...
      private ClassA m_classaObject;
      private ClassB m_classbObject;
}
```

8. Proper Use of Constants

No piece of code should contain any numeral or string constants other than 0 or 1. All other constants should be explicitly declared in the constants section of that class. This has the following advantages:

- Increases the readability of code.
- Makes modifications much easier as the maintainer has to just concentrate on the constants section of the class instead of scourging the entire source code trying to find the various constants embedded in it.
- Helps understand the various assumptions the author is using, making it easier to debug the application when needed.

Example

Avoid this:

```
public void printMyStandardErrorMessage()
{
            System.out.println(  "wow!  Something  went  wrong!  I'm
      out" );
            return;
}
```

Instead use:

```
public void printMyStandardErrorMessage()
{
            System.out.println( S_ERROR );
            return;
}
```

where `S_ERROR` has been declared as the following in the class constants section.:

```
public final String S_ERROR = "wow! Something went wrong! I'm
out";
```

9. Calling Methods

In an object-oriented language, every operation that is performed has to be performed on some object. Thus, it is advisable to write code where the object on which the operation is being performed is clearly specified. So avoid the following code pattern:

```
public class MyCoolClass

{
        ...
        ...
        public void doSomethingCool()
        {
                ...
                doSomethingReallyCool();
                ...
        }
        ...
        ...
        private void doSoemthingReallyCool()
        {
                ...
                ...
        }
}
```

instead use the following as it clearly points out the object on which the method `doSomethingReallyCool()` is being performed.

```
public class MyCoolClass
{
        ...
        ...
        public void doSomethingCool()
        {
                ...
                this.doSomethingReallyCool();
                ...
        }
        ...
        ...
        private void doSoemthingReallyCool()
        {
                ...
                ...
        }
}
```

This is even more useful when we are dealing with *anonymous* or inner Java classes. For example consider the following piece of code:

```
public class MainFrame extends Jframe

{
      public MainFrame()
      {
            super( S_TITLE );
            ...
            this.addWindowListner( new WindowAdaptor()
                  {
                        public void windowClosing( WindowEvent e )
                        {
                              onEventClosing( e );
                              return;
                        }
                        ...
                  }
            );
            ...
      }
}
```

Here it is not clear whether the method `onEventClosing()` is being called on the anonymous class or the `MainFrame` class. Of course the complier will first try calling this method on the anonymous class and if it does not exist then it will call it on the `MainFrame` class. But now if suddenly someone introduced an `onEventClosing()` method in the anonymous class, things could go terribly wrong. Thus, the following pattern should be used instead:

```
public class MainFrame extends Jframe
{
      public MainFrame()
      {
            super( S_TITLE );
            ...
            this.addActionListner( new WindowAdaptor()
                  {
                        public void windowClosing( WindowEvent e )
                        {
                              MainFrame.this.onEventClosing( e );
                              return;
                        }
                        ...
                  }
            );
            ...
      }
}
```

Now it is very clear that the `onEventClosing()` method is being called on the class `MainFrame` and not on the inner anonymous class.

10. Conclusion

This article is an attempt at standardizing the coding practices in the CelLEST project group. It is possible that following all these recommendations might initially slow one down a notch or two, but it would be well worth the effort in the long run as:

1. It would lead to more readable and thus, less buggy programs,
2. It would really make the life of the person who inherits your code much easier.

Thus, it is hoped that adoption of standardized coding practices would eventually lead to a better understanding and cooperation in the entire project group.

11. Example of A Java File That Follows These Guidelines

The following is an example of a real life Java class that has been coded in accordance with the guidelines listed in this paper. This might serve as a good example of how to actually incorporate the suggestions listed in this article into your own source code.

```
import java.io.*;
import java.lang.*;
import java.util.*;

/**
 * This class is a simple struct for encapsulating fields
 * and their postions in the screen file! Its is intended
 * to be used only by the class AutoTemplate
 * <p>
 * Created: Tue Jun 13 19:08:05 2000
 * <p>
 * @author Rohit Kapoor
 * @version 1.0
 *
 */

class FieldPosStruct
{


//////////////////////////////////////////////////////
// constructors
//////////////////////////////////////////////////////

      /**
       * The default constructor. Use create with this.
       *
       */
      public FieldPosStruct()
      {
            m_sFieldName = null;
            m_nPos = 0;

            return;
      }

      /**
       * The copy constructor.
       * <p>
       * @param fps The object to be copied
       *
```

```
     */
    public FieldPosStruct( final FieldPosStruct fps )
    {
        if ( fps.m_sFieldName != null )
            m_sFieldName = new String( fps.m_sFieldName );
        else
            m_sFieldName = null;

        m_nPos = fps.m_nPos;

        return;
    }


    /**
     * This is the full featured constructor.
     * <p>
     * @param sFieldName The field name
     * @param nPos Its position
     *
     */
    public FieldPosStruct( final String sFieldName,
        final int nPos )
    {
        this.create( sFieldName, nPos );

        return;
    }


////////////////////////////////////////////////////
// operations (instance functions)
////////////////////////////////////////////////////

    /**
     * Use this when this class has been created using the
     * default constructor.
     * <p>
     * @param sFieldName The field name
     * @param nPos Its position
     *
     */
    public void create( final String sFieldName,
        final int nPos )
    {
        m_sFieldName = new String( sFieldName );
        m_nPos = nPos;

        return;
    }
```

```
/**
 * Accessor for field name
 * <p>
 * @return The field name
 *
 */
public String getFieldName()
{
     return new String( m_sFieldName );
}

/**
 * Accessor for field position
 * <p>
 * @return The field position
 *
 */
public int getFieldPos()
{
     return m_nPos;
}

/**
 * Set function for field name
 * <p>
 * @param sFieldName The field name
 *
 */
public void setFieldName( final String sFieldName )
{
     if ( sFieldName == null )
          m_sFieldName = null;
     else
          m_sFieldName = new String( sFieldName );

     return;
}

/**
 * Set function for field position
 * <p>
 * @param nPos The field position
 *
 */
public void setFieldPos( final int nPos )
{
     m_nPos = nPos;

     return;
}
```

```
/**
 * This is the standard <code>equals</code> function. This
 * one returns <code>true</code> only if the object being
 * passed is not <code>null</code> and it is an instance of
 * FieldPosStruct class and its internal fields have the
 * same contents.
 * <p>
 * @param theObject The object being tested for equality
 * <p>
 * @return <code>true</code> if the two objects are equal
 * else it returns <code>false</code>
 *
 */
public boolean equals( Object theObject )
{
      //
      // null check!
      //
      if ( theObject == null )
            return false;

      //
      // test the simplest case
      //
      if ( this == theObject )
            return true;

      //
      // check that the fields are equal
      //
      if ( theObject instanceof FieldPosStruct )
      {
            FieldPosStruct fpsThis =
                  ( FieldPosStruct )theObject;

            boolean fFieldsEqual = true;
            if ( ( m_sFieldName == null ) && (
                  fpsThis.m_sFieldName != null ) )
                  fFieldsEqual = false;

            if ( ( m_sFieldName != null ) && (
                  !m_sFieldName.equals( fpsThis.m_sFieldName
                  ) ) )
                  fFieldsEqual = false;

            if ( m_nPos != fpsThis.m_nPos )
                  fFieldsEqual = false;

            return fFieldsEqual;
      }
```

```
           return false;
      }


////////////////////////////////////////////////////
// implementation (instance variables)
////////////////////////////////////////////////////

      /** @serial Holds the field name */
      private String m_sFieldName;

      /** @serial Holds the field position */
      private int m_nPos;
}
```